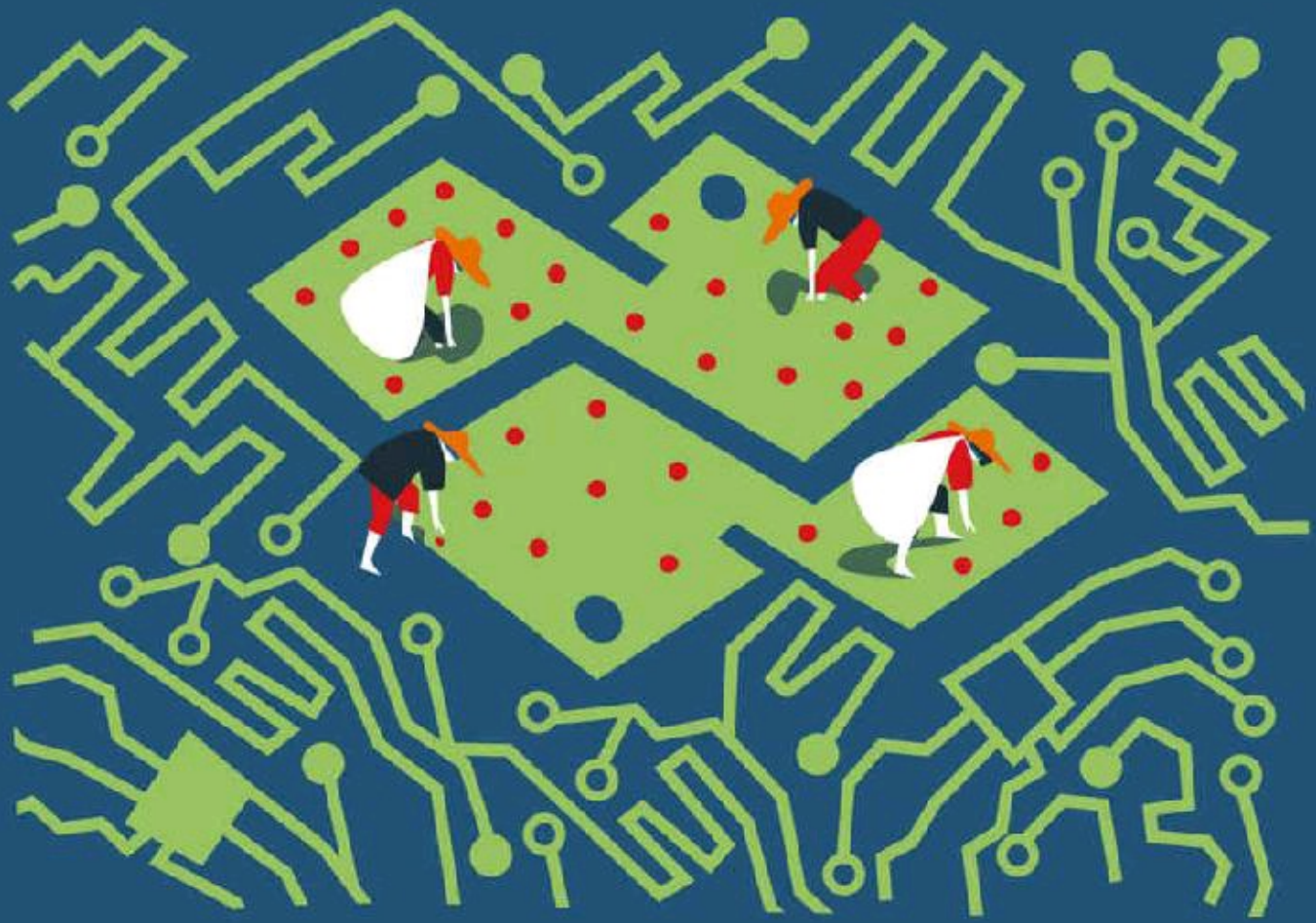
SEBASTIAN RASCHKA

Machine Learning con Python



Costruire algoritmi per generare conoscenza

APOGEO

Machine Learning con Python
Costruire algoritmi per generare conoscenza

Sebastian Raschka

© Apogeo - IF - Idee editoriali Feltrinelli s.r.l.
Socio Unico Giangiacomo Feltrinelli Editore s.r.l.

ISBN edizione cartacea: 9788850333974

Copyright © Packt Publishing 2015. First published in the English language under the title “Python Machine Learning” – (9781783555130).

Il presente file può essere usato esclusivamente per finalità di carattere personale. Tutti i contenuti sono protetti dalla Legge sul diritto d’autore.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

L’edizione cartacea è in vendita nelle migliori librerie.

~

Sito web: www.apogeonline.com

Scopri le novità di Apogeo su Facebook

Seguici su Twitter @apogeonline

Collegati con noi su LinkedIn

Rimani aggiornato iscrivendoti alla nostra newsletter

Viviamo sommersi da un vero e proprio diluvio di dati. Secondo recenti stime, quotidianamente vengono generati 2.5 quintilioni di dati. Quest'enorme massa di dati è stata generata per oltre il 90% nel corso dell'ultimo decennio. Sfortunatamente, la maggior parte di queste informazioni non può essere utilizzata direttamente da noi esseri umani. Questo per il fatto che tali dati vanno ben oltre i mezzi dei normali metodi analitici oppure, semplicemente, superano le nostre limitate capacità di comprensione. Tramite il *machine learning*, l'apprendimento automatico, incarichiamo i computer di elaborare, imparare e trarre conoscenze utili da questa altrimenti impenetrabile montagna di dati. A partire dai grossi supercomputer che supportano i motori di ricerca di Google fino agli smartphone che tutti noi abbiamo in tasca, contiamo costantemente sugli algoritmi di apprendimento automatico per cercare di muoverci nel mondo che ci circonda, spesso senza neppure accorgercene.

Quasi fossimo moderni pionieri, in questo mondo inesplorato costellato di montagne di dati, abbiamo pertanto il compito di conoscere meglio le tecniche di apprendimento automatico. Che cosa si intende con machine learning e come funziona? Come possiamo utilizzare il machine learning per gettare luce sull'ignoto, per dare nuovo impulso a un'attività o, semplicemente, per scoprire che cosa ne pensa Internet del nostro film preferito? Tutto questo e molto altro ancora verrà trattato nei capitoli che seguono, scritti dal mio amico e collega Sebastian Raschka.

Quando non è impegnato a tenere a bada il mio altrimenti irascibile cane, Sebastian dedica instancabilmente il proprio tempo libero alla comunità del machine learning open source. Nel corso degli ultimi anni, Sebastian ha sviluppato decine di guide su argomenti che vanno dal machine learning alla visualizzazione dei dati in Python. Inoltre, ha realizzato o contribuito allo sviluppo di numerosi pacchetti open source realizzati in Python, molti dei quali fanno ora parte del normale flusso di lavoro di Python nell'ambito del machine learning.

Grazie alla sua enorme esperienza in questo campo, sono sicuro che i suoi suggerimenti nell'ambito della programmazione in Python rivolta al machine

learning risulteranno preziosi per tutti coloro che operano a qualsiasi livello di esperienza. Raccomando di cuore questo libro a chiunque voglia acquisire una comprensione più ampia e pratica delle tecniche del machine learning.

Randal S. Olson
Ricercatore nei campi dell'intelligenza artificiale e del machine learning
Università della Pennsylvania

Probabilmente non ho bisogno di dirvi che il machine learning è una delle tecnologie più stimolanti della nostra epoca. Tutte le grandi aziende, fra cui Google, Facebook, Apple, Amazon, IBM e molte altre ancora investono grandi somme nella ricerca e nelle applicazioni nell'ambito del machine learning e questo per ottimi motivi. Anche se può sembrare che il "machine learning" sia uno degli elementi chiave per comprendere i nostri tempi, certamente il termine non è sulla bocca di tutti. Questo interessante campo apre la via a nuove possibilità ed è diventato un elemento indispensabile nella nostra vita quotidiana. Interviene ogni volta che parliamo con l'assistente vocale dello smartphone, che un sito invia consigli per gli acquisti ai clienti, viene bloccata una truffa su una carta di credito, viene eliminato un messaggio di spam dalle caselle di posta elettronica, viene diagnosticata una malattia. L'elenco sarebbe potrebbe continuare a lungo.

Se volete occuparvi di machine learning, se volete imparare a risolvere al meglio i problemi o anche considerare una carriera nell'ambito del machine learning, questo libro fa proprio per voi, ma per chi è alle prime armi, i concetti teorici su cui si basano gli algoritmi di machine learning potrebbero risultare piuttosto ostici. Tuttavia, nel corso degli ultimi anni sono stati pubblicati ottimi libri pratici, utili per iniziare la pratica del machine learning implementando potenti algoritmi di apprendimento. È mia opinione che la presenza di esempi pratici di codice sia fondamentale. Il codice presentato illustra i concetti, consentendo di mettere all'opera il materiale appreso. Tuttavia, considerate anche il fatto che: da un grande potere derivano grandi responsabilità! I concetti su cui si basa il machine learning sono troppo importanti per rimanere nascosti in una scatola. La mia personale missione è quella di fornirvi un libro differente, un libro nel quale troverete tutti i dettagli relativi ai concetti su cui si basa il machine learning e che offre spiegazioni intuitive e rigorose sul funzionamento degli algoritmi di machine learning, sul loro uso e, soprattutto, sul modo in cui evitare gli errori più comuni.

Se digitate i termini "machine learning" in Google Scholar, otterrete l'impressionante numero di 3.600.000 pubblicazioni. Naturalmente, potremmo anche trattare ogni singolo dettaglio dei vari algoritmi e delle relative applicazioni

che sono emersi nel corso degli ultimi sessant'anni. Tuttavia, in questo libro, ci imbarcheremo in un interessante viaggio che tratterà gli argomenti e i concetti essenziali necessari per iniziare a comprendere questa branca dell'informatica. Se sentite che la vostra sete di conoscenza non è ancora soddisfatta, vi troverete molte altre risorse utili, che potrete utilizzare per approfondire questi argomenti.

Se avete già alle spalle qualche studio teorico nel campo del machine learning, questo libro vi aiuterà a mettere in pratica le vostre conoscenze. Se avete già impiegato tecniche di machine learning e volete capire qualcosa di più sul modo in cui funzionano tali algoritmi, questo libro per voi! Ma non preoccupatevi anche se siete completamente all'oscuro dell'argomento: avete ancora più ragioni per interessarvi. Vi prometto che il machine learning cambierà il modo in cui rifletterete sulla soluzione dei problemi e vi mostrerà come affrontarli, sfruttando tutte le potenzialità insite nei dati.

Prima di addentrarci nel campo del machine learning, cominciamo a rispondere alla domanda più importante. Perché Python?

La risposta è semplice: si tratta di un linguaggio potente e anche accessibile. Python è diventato il linguaggio di programmazione più diffuso per l'elaborazione dei dati, poiché consente di trascurare tutta la parte noiosa della programmazione, offrendo un ambiente in cui è possibile mettere in pratica le idee e vedere in azione i concetti.

Riflettendo sul mio personale percorso professionale, posso davvero dire che lo studio del machine learning mi ha reso migliore come scienziato, pensatore e risolutore di problemi. In questo libro vorrei condividere questa conoscenza con voi. La conoscenza si acquisisce con l'apprendimento e la chiave per farlo è il nostro entusiasmo, ma la padronanza dei concetti può arrivare solo con la pratica. La strada che avete davanti può essere talvolta irta di ostacoli e alcuni argomenti possono essere più complessi di altri, ma spero che cogliate questa opportunità e vi concentriate sulla ricca ricompensa. Ricordate che in questo viaggio siamo imbarcati insieme: nel corso di questo libro aggiungerò al vostro arsenale tante tecniche potenti, che vi aiuteranno a risolvere anche i problemi più difficili, sulla base dei dati disponibili.

Struttura del libro

Il Capitolo 1, *Dare ai computer la capacità di apprendere dai dati*, introduce i principali aspetti dell'uso di tecniche di machine learning per affrontare vari tipi di problemi. Inoltre, si occupa dei passi essenziali per la creazione di un tipico modello di machine learning, costruendo la catena (pipeline) che vi guiderà nel corso dei capitoli successivi.

Il Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, torna alle origini del machine learning e introduce i classificatori binari perceptron e i neuroni lineari adattativi. Questo capitolo rappresenta una semplice introduzione alle basi della classificazione e si concentra sull'interazione esistente fra gli algoritmi di ottimizzazione e il machine learning.

Il Capitolo 3, *I classificatori di machine learning di scikit-learn*, descrive i principali algoritmi di machine learning rivolti alla classificazione, fornendo esempi pratici che sfruttano le funzionalità di una delle più note e ricche librerie di machine learning open source: scikit-learn.

Il Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione*, insegna a gestire i più comuni problemi nei dataset grezzi, per esempio i dati mancanti. Inoltre discute vari approcci per identificare le caratteristiche più informative nei dataset e insegna a preparare le variabili di vari tipi, in modo che rappresentino input corretti per gli algoritmi di machine learning.

Il Capitolo 5, *Compressione dei dati tramite la riduzione della dimensionalità*, descrive le tecniche essenziali per ridurre il numero di caratteristiche di un dataset a insiemi più piccoli, mantenendo la maggior parte delle informazioni utili e discriminanti. Discute l'approccio standard alla riduzione della dimensionalità tramite l'analisi del componente principale, confrontandolo con le tecniche di trasformazione con supervisione e non lineari.

Il Capitolo 6, *Valutazione dei modelli e ottimizzazione degli iperparametri*, discute i pro e i contro della stima delle prestazioni nei modelli predittivi. Inoltre discute varie metriche per la misurazione delle prestazioni dei modelli e le tecniche di ottimizzazione degli algoritmi di machine learning.

Il Capitolo 7, *Combinare più modelli: l'apprendimento d'insieme*, introduce vari concetti che prevedono di combinare efficacemente più algoritmi di apprendimento. Insegna a realizzare insiemi di "esperti" per superare i punti deboli dei singoli algoritmi, ottenendo previsioni più accurate e affidabili.

Il Capitolo 8, *Tecniche di machine learning per l'analisi del sentiment*, si occupa dei passi essenziali necessari per trasformare i dati testuali in rappresentazioni significative tramite algoritmi di machine learning, con lo scopo di prevedere l'opinione del pubblico sulla base dei testi che produce.

Il Capitolo 9, *Embedding di un modello in un'applicazione web*, prosegue la discussione sul modello predittivo del capitolo precedente e accompagna attraverso i passi essenziali dello sviluppo di applicazioni web con modelli di machine learning embedded.

Il Capitolo 10, *Previsioni di variabili target continue: l'analisi a regressione*, discute le tecniche essenziali di modellazione delle relazioni lineari fra variabili target e variabili di risposta, per effettuare previsioni su una scala continua. Dopo aver introdotto vari modelli lineari, parla di regressione polinomiale e approcci basati su una struttura ad albero.

Il Capitolo 11, *Lavorare con dati senza etichette: l'analisi a cluster*, sposta l'attenzione su una branca diversa del machine learning: l'apprendimento senza supervisione. Applichiamo gli algoritmi di tre diverse famiglie di algoritmi a clustering, per trovare gruppi di oggetti che condividono un certo grado di similarità.

Il Capitolo 12, *Reti neurali artificiali per il riconoscimento delle immagini*, estende i concetti dell'ottimizzazione basata su gradienti che abbiamo introdotto nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, per realizzare potenti reti neurali multilivello, sulla base di algoritmi a retropropagazione.

Il Capitolo 13, *Parallelizzare l'addestramento delle reti neurali con Theano*, estende le conoscenze trattate nel capitolo precedente offrendo una guida pratica all'addestramento efficiente di reti neurali. L'argomento del capitolo è principalmente Theano, una libreria open source Python che consente di utilizzare i core delle moderne GPU.

Dotazione software necessaria

L'esecuzione degli esempi di codice forniti in questo libro richiede l'installazione di Python 3.4.3 o una versione successiva su un sistema Mac OS X, Linux o Microsoft Windows. Faremo frequentemente uso delle librerie essenziali di Python dedicate al calcolo scientifico, fra cui SciPy, NumPy, scikit-learn, matplotlib e pandas.

Il primo capitolo fornirà tutte le istruzioni e i suggerimenti necessari per configurare l'ambiente Python e queste librerie di base. Successivamente aggiungeremo ulteriori librerie al nostro repertorio, e le relative istruzioni di installazione verranno fornite nei capitoli successivi: la libreria NLTK, dedicata all'elaborazione del linguaggio naturale (Capitolo 8, *Tecniche di machine learning per l'analisi del sentiment*), il framework web Flask (Capitolo 9, *Embedding di un modello in un'applicazione web*), la libreria seaborn per la visualizzazione dei dati statistici (Capitolo 10, *Previsioni di variabili target continue: l'analisi a regressione*) e Theano per l'addestramento efficiente di reti neurali su unità GPU (Capitolo 13, *Parallelizzare l'addestramento delle reti neurali con Theano*).

A chi è rivolto questo libro

A chi intende scoprire come utilizzare Python per iniziare a rispondere a domande critiche basate sui dati. Che vogliate iniziare da zero o vogliate estendere le vostre conoscenze nell'ambito dell'elaborazione dei dati, questa è una risorsa essenziale e imperdibile.

Convenzioni

In questo libro, troverete vari stili di testo per distinguere diversi tipi di informazioni. Ecco alcuni esempi degli stili utilizzati e del loro significato.

Gli elementi di codice nel testo, i nomi di tabelle di un database, i nomi di cartelle, i nomi di file, le estensioni di file, i percorsi, gli indirizzi URL, gli input da inserire sono tutti scritti utilizzando il carattere monospaziato, per esempio: "I pacchetti già installati possono essere aggiornati con il flag `--upgrade`".

Un blocco di codice ha il seguente aspetto:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> X = df.iloc[0:100, [0, 2]].values
>>> plt.scatter(X[:50, 0], X[:50, 1],
...             color='red', marker='x', label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...             color='blue', marker='o', label='versicolor')
>>> plt.xlabel('sepal length')
>>> plt.ylabel('petal length')
>>> plt.legend(loc='upper left')

>>> plt.show()
```

Ogni input da inserire nella riga di comando e ogni output è scritto nel seguente modo:

```
> dot -Tpng tree.dot -o tree.png
```

I nuovi termini e le parole chiave di un discorso sono indicate in *corsivo*. Le parole che compaiono sullo schermo, per esempio nei menu o nelle finestre di dialogo, sono sempre indicate in *corsivo*.

NOTA

Avvertimenti e note importanti sono evidenziate in questo modo.

Approfondimento

Approfondimenti e chiarificazioni ai concetti e agli argomenti trattati nel corso del testo vengono mostrati in questo modo.

Scarica i file degli esempi

L'autore ha creato un repository su GitHub per mettere ha disposizione dei lettori il codice degli esempi e i dataset utlizzati nel libro. L'indirizzo è: https://github.com/rasbt/python-machine-learning-book.

L’autore

Sebastian Raschka, dottorando presso la Michigan State University, sviluppa modelli di calcolo computerizzato applicato alla biologia. È stato valutato da Analytics Vidhya come lo scienziato più influente nell’ambito della *data science* su GitHub. Vanta un’esperienza pluriennale di programmazione in Python e ha tenuto diversi seminari sulle applicazioni pratiche della scienza dei dati e sull’apprendimento automatico. Il fatto di parlare e scrivere di scienza dei dati, machine learning e Python lo ha motivato nella scrittura di questo libro per aiutare anche altri a sviluppare soluzioni basate sui dati, rivolgendosi anche a coloro che non sono dotati di un background specifico in questi ambiti.

Ha anche contribuito attivamente allo sviluppo di progetti e metodi open source che ha implementato e che ora vengono utilizzati con successo nelle competizioni di machine learning, come Kaggle. Nel tempo libero, progetta modelli per le previsioni dei risultati sportivi e quando non è davanti a un computer, si dedica allo sport.

I revisori

Richard Dutton ha iniziato a programmare su un Sinclair ZX Spectrum all'età di otto anni e le sue ossessioni lo hanno trasportato attraverso una serie confusa di tecnologie e incarichi nei campi della tecnologia e della finanza.

Ha lavorato con Microsoft e come direttore in Barclays, ma la sua attuale ossessione è un mix di Python, machine learning e concatenamento a blocchi.

Quando non è seduto a un computer, lo si può trovare in palestra o a casa, dietro a un calice di vino mentre osserva il display del suo iPhone. Lui chiama tutto questo "equilibrio".

Dave Julian è consulente informatico e docente con oltre quindici anni di esperienza. Ha lavorato come tecnico, project manager, programmatore e sviluppatore web. I suoi attuali progetti comprendono lo sviluppo di uno strumento di analisi della serricoltura nell'ambito delle strategie di lotta integrata ai parassiti. È particolarmente interessato alle interazioni fra biologia e tecnologie, con una forte convinzione che, in futuro, macchine intelligenti potranno aiutare a risolvere i più grandi problemi che affliggono il mondo.

Vahid Mirjalili ha conseguito la laurea in ingegneria meccanica presso la Michigan State University, dove ha sviluppato innovative tecniche per il raffinamento della struttura proteica, utilizzando simulazioni molecolari dinamiche. Combinando la sua conoscenza nei campi della statistica, del data mining e della fisica ha sviluppato potenti approcci basati sui dati che hanno aiutato lui e il suo team di ricercatori a vincere due recenti competizioni a livello mondiale sulla predizione e il raffinamento della struttura proteica, CASP, nel 2012 e nel 2014.

Nel corso del suo dottorato, ha deciso di unirsi al Computer Science and Engineering Department della Michigan State University, per specializzarsi nel campo dell'apprendimento automatico. I suoi attuali progetti di ricerca riguardano lo sviluppo di algoritmi di machine learning senza supervisione per il mining di grossi data set. È anche un appassionato programmatore Python e condivide le sue implementazioni degli algoritmi di clustering sul suo sito web personale, http://vahidmirjalili.com.

Hamidreza Sattari è un professionista informatico che è stato coinvolto in varie aree dell'ingegneria software, dalla programmazione all'architettura, fino alla amministrazione. È dotato di un master in ingegneria del software conseguito presso la Herriot-Watt University, in Gran Bretagna e di una laurea in ingegneria elettronica conseguita presso la Azad University di Teheran, in Iran. Negli ultimi anni, ha rivolto la sua attenzione all'elaborazione di grosse quantità di dati e all'apprendimento automatico. È coautore del libro *Spring Web Services 2 Cookbook* e cura il blog http://justdeveloped-blog.blogspot.com.

Dmytro Taranovsky è un ingegnere software che manifesta un grande interesse e background nell'ambito di Python, Linux e del machine learning. Originario di Kiev, Ucraina, si è trasferito negli Stati Uniti nel 1996. Fin dalla più tenera età, ha dimostrato una passione per la scienza e la conoscenza, vincendo competizioni in campo matematico e fisico. Nel 1999 è stato scelto quale membro del U.S. Physics Team. Nel 2005 si è laureato presso il Massachusetts Institute of Technology, specializzandosi in Matematica. Successivamente ha lavorato come ingegnere software su un sistema di trasformazione del testo per trascrizioni mediche assistite al computer (eScription). Anche se originariamente ha lavorato in Perl, apprezza la potenza e la chiarezza di Python ed è stato in grado di scalare il proprio sistema in modo da farlo operare su grandi quantità di dati. Successivamente ha lavorato come ingegnere software e analista per una società di trading. Ha fornito significativi contributi alle basi matematiche, compresa la creazione e lo sviluppo di un'estensione al linguaggio della teoria degli insiemi e alla sua connessione con gli assiomi a grande cardinalità, sviluppando un concetto di verità costruttiva e creando un sistema di notazione ordinale e poi implementando il tutto in Python. Gli piace anche leggere, andare a spasso e tenta di rendere il mondo un luogo migliore.

Intendo ringraziare i miei docenti, Arun Ross e Pang-Ning Tan e tutti coloro che mi hanno ispirato e che hanno acceso in me un grande interesse nella classificazione, nel machine learning e nel data mining.

Vorrei approfittare di questa opportunità per ringraziare la grande comunità di Python e gli sviluppatori di pacchetti open source che mi hanno aiutato a creare l'ambiente perfetto per la ricerca scientifica e la scienza dei dati.

Un ringraziamento speciale va agli sviluppatori di scikit-learn. Avendo contribuito al progetto, ho avuto il piacere di lavorare con ottime persone, che non solo sono estremamente competenti nell'ambito del machine learning, ma sono anche eccellenti programmatori.

Infine vorrei ringraziare voi per aver dimostrato interesse su questo libro e spero sinceramente di poter trasmettere il mio entusiasmo, per esortarvi a unirvi alla grande comunità di Python e del machine learning.

Dare ai computer la capacità di apprendere dai dati

È mia opinione che il *machine learning*, o apprendimento automatico, ovvero l'applicazione e lo studio degli algoritmi che estraggono informazioni utili dalla massa informe dei dati, sia il campo più interessante dell'informatica. Viviamo in un'era in cui i dati sono disponibili in una quantità sovrabbondante; utilizzando gli algoritmi di autoapprendimento nel campo del machine learning, siamo in grado di trasformare questi dati in *conoscenza*. Grazie alle tante e sempre più potenti librerie open source che sono state sviluppate nel corso degli ultimi anni, non vi è probabilmente mai stato un momento migliore per occuparsi di machine learning e per imparare a utilizzare questi potenti algoritmi per individuare schemi nei dati ed eseguire previsioni a proposito degli eventi futuri.

In questo capitolo parleremo dei concetti principali e dei vari tipi di machine learning. Dopo un'introduzione alla terminologia di base, getteremo le basi per l'utilizzo delle tecniche di machine learning per la risoluzione di problemi pratici.

In questo capitolo, affronteremo i seguenti argomenti.

- Concetti generali di machine learning.
- I tre tipi di apprendimento e la terminologia di base.
- Gli elementi costitutivi per la realizzazione di sistemi di machine learning.
- Installazione e configurazione di Python per l'analisi dei dati e il machine learning.

Costruire macchine intelligenti per trasformare i dati in conoscenza

In questa nuova era di tecnologie, vi è una risorsa che abbiamo disponibile in grande abbondanza: abbiamo una grande quantità di dati strutturati e non strutturati. Nella seconda metà del ventesimo secolo, il machine learning si è evoluto come una branca dell'*intelligenza artificiale*, che prevede lo sviluppo di algoritmi di autoapprendimento in grado di acquisire conoscenze dai dati, con lo scopo di effettuare previsioni. Invece di richiedere una presenza umana, in grado di individuare manualmente delle regole e costruire dei modelli per l'analisi di grandi quantità di dati, il machine learning offre un'alternativa più efficiente per catturare la conoscenza insita nei dati, col fine di migliorare gradualmente le prestazioni dei modelli previsionali e poi prendere decisioni guidate dai dati. Non solo il machine learning sta diventando sempre più importante nel campo della ricerca basata su computer, ma gioca anche un ruolo sempre più preminente nella nostra vita quotidiana. Grazie al machine learning, possiamo contare su solidi filtri in grado di eliminare lo spam dalla nostra posta elettronica, su comodi software per il riconoscimento del testo e della voce, su affidabili motori di ricerca per il Web, su giocatori artificiali di scacchi sempre più abili e, speriamo al più presto, su automobili a guida automatica, che siano sicure ed efficienti.

I tre diversi tipi di machine learning

In questo paragrafo esamineremo i tre diversi tipi di machine learning: *apprendimento con supervisione*, *apprendimento senza supervisione* e *apprendimento di rafforzamento*. Esamineremo le differenze fondamentali che esistono fra questi tre diversi tipi di apprendimento (Figura 1.1) e, utilizzando esempi concettuali, svilupperemo alcune idee relative ai domini pratici dei problemi ai quali tali tecniche possono essere applicate.

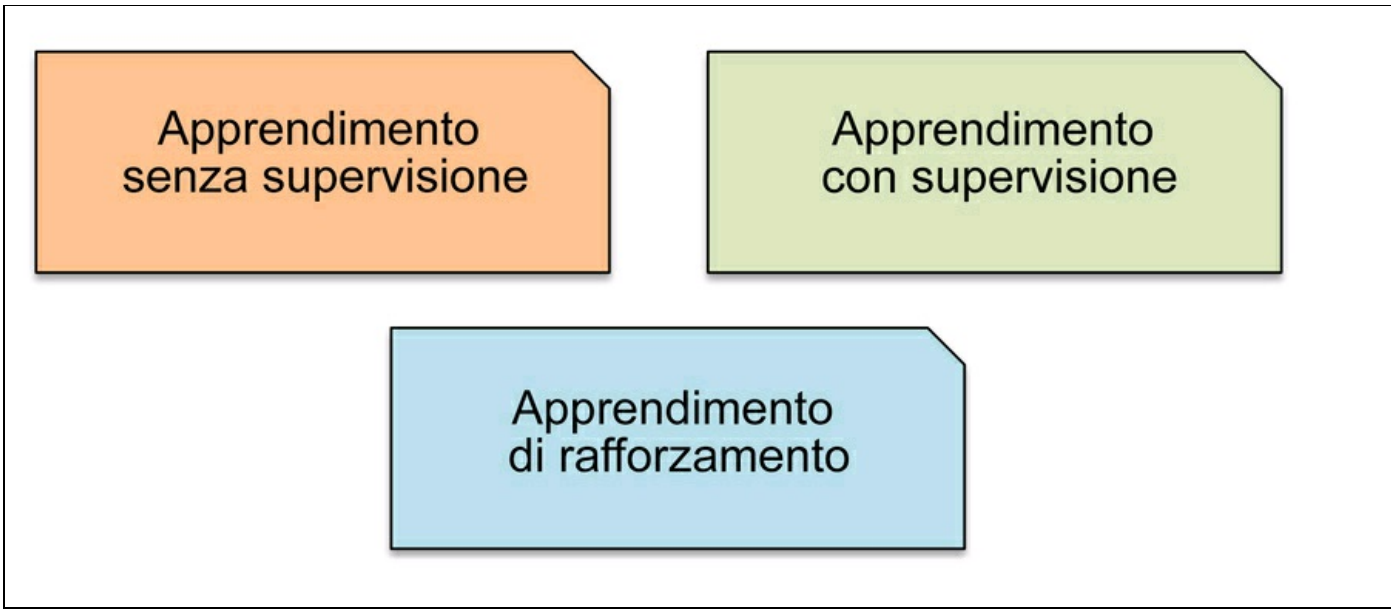


Figura 1.1

Effettuare previsioni sul futuro grazie all'apprendimento con supervisione

Lo scopo principale dell'apprendimento con supervisione consiste nel trarre un modello a partire da dati di *addestramento* etichettati, i quali ci consentono di effettuare previsioni relative a dati non disponibili o futuri (Figura 1.2). Qui il termine *con supervisione* fa riferimento al fatto che nell'insieme di campioni i segnali di output desiderati (le etichette) sono già noti.

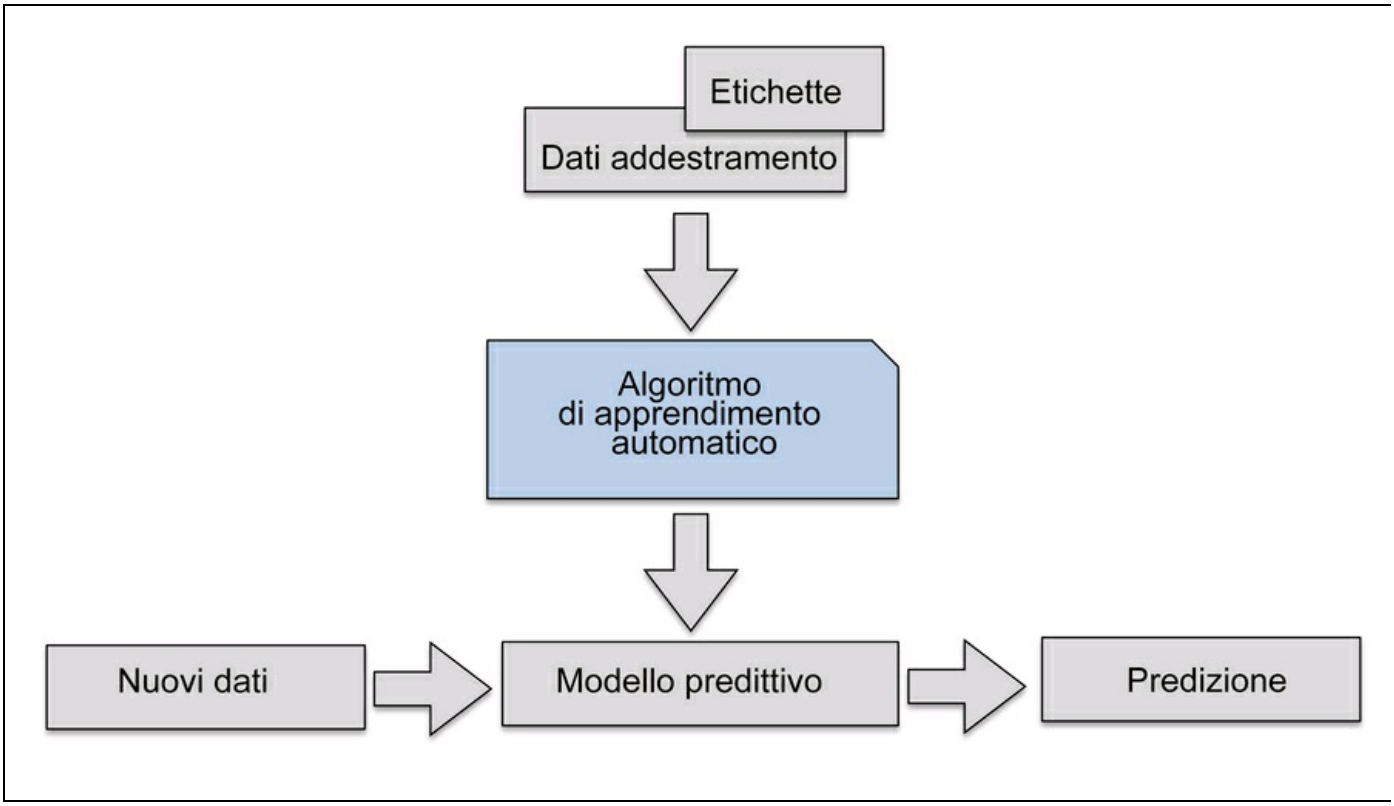


Figura 1.2

Considerando l'esempio del filtraggio dei messaggi spam di posta elettronica, possiamo addestrare un modello applicando un algoritmo di apprendimento con supervisione a un insieme di messaggi di posta elettronica già etichettati, che siano stati correttamente contrassegnati come spam oppure non-spam, per fargli determinare se un nuovo messaggio di posta elettronica appartiene all'una o all'altra categoria. Un compito di apprendimento con supervisione, sulla base di *etichette delle classi* discrete, come nel precedente esempio del filtraggio dello spam nella posta elettronica, è chiamato anche compito di *classificazione*. Un'altra sottocategoria di apprendimento con supervisione è la *regressione*, dove il segnale risultante è un valore continuo.

Classificazione per la predizione delle etichette delle classi

La classificazione è una sottocategoria dell'apprendimento con supervisione, dove l'obiettivo è quello di prevedere le etichette di categoria delle classi per le nuove istanze, sulla base delle osservazioni compiute nel passato. Queste etichette sono valori discreti, non ordinati, che possono essere considerati come *appartenenti a un gruppo* delle istanze. L'esempio menzionato in precedenza del rilevamento dello spam nella posta elettronica rappresenta un tipico esempio di *classificazione binaria*: l'algoritmo di apprendimento automatico impara un insieme di regole con

lo scopo di distinguere fra due possibili classi: messaggi di posta elettronica che sono spam o che sono non-spam.

Tuttavia, l'insieme delle etichette delle classi non deve necessariamente avere una natura binaria. Il modello predittivo individuato da un algoritmo di apprendimento con supervisione può considerare ogni etichetta della classe che sia presente nel dataset di apprendimento di una nuova istanza che non sia dotata di etichetta. Un tipico esempio di un compito di *classificazione multiclasse* è il riconoscimento del testo scritto a mano. Qui possiamo raccogliere un dataset di apprendimento che è costituito da più esempi di scrittura a mano di ciascuna lettera dell'alfabeto. Ora, se un utente fornisce un nuovo carattere scritto a mano tramite un dispositivo di input, il nostro modello predittivo sarà in grado di prevedere con una certa precisione la lettera corretta dell'alfabeto. Tuttavia, il nostro sistema di apprendimento automatico non sarebbe in grado di riconoscere correttamente le cifre da 0 a 9, per esempio, se queste non facevano già parte del dataset di apprendimento.

La Figura 1.3 illustra il concetto del compito di classificazione binaria sulla base di campioni di apprendimento: quindici di essi sono etichettati come *classe negativa* (i cerchi) e altrettanti campioni sono etichettati come *classe positiva* (i segni +). In questa situazione, il nostro dataset è bidimensionale, il che significa che a ogni campione possono essere associati due colori: x_1 e x_2. Ora, possiamo utilizzare un algoritmo di apprendimento automatico con supervisione per trarre una regola (il confine decisionale è rappresentato dalla linea tratteggiata) che sia in grado di separare queste due classi e classificare i nuovi dati in ognuna di queste due categorie sulla base dei loro valori x_1 e x_2.

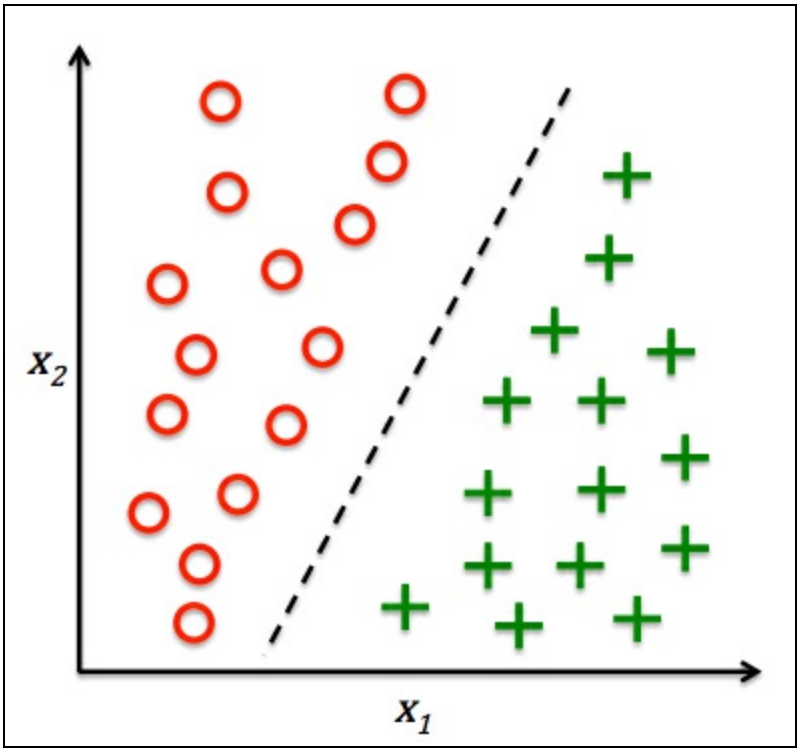


Figura 1.3

Regressione per la previsione di risultati continui

Nel paragrafo precedente abbiamo visto che il compito di classificazione consiste nell'assegnare alle istanze etichette di categorie non ordinate. Un secondo tipo di apprendimento con supervisione è la previsione di risultati continui, chiamata anche analisi di regressione. Nell'*analisi di regressione*, abbiamo un certo numero di variabili predittive (descrittive) e una variabile target continua (risultato): cerchiamo di trovare una relazione fra queste variabili, tale che ci consenta di prevedere un risultato.

Per esempio, supponiamo di essere interessati a prevedere le valutazioni di una prova scritta dei nostri studenti. Se vi è una relazione fra il tempo dedicato a studiare per il test e i risultati finali, potremmo utilizzare questi tempi come dati di apprendimento per derivare un modello che utilizzi proprio il tempo dedicato allo studio per prevedere le valutazioni dei futuri studenti che pensano di svolgere questo test.

NOTA

Il termine *regressione* è stato coniato da Francis Galton nel suo articolo *Regression Towards Mediocrity in Hereditary Stature*, scritto nel 1886. Galton ha descritto un fenomeno biologico in base al quale la varianza di *altezza* in una popolazione non aumenta nel corso del tempo. Ha osservato che la statura dei genitori non viene trasmessa ai figli, ma regredisce nella direzione della media della popolazione.

La Figura 1.4 illustra il concetto di *regressione lineare*. Data una variabile predittiva *x* e una variabile risposta *y*, tracciamo una linea retta attraverso questi dati in modo da minimizzare la distanza (si parla infatti di distanza quadratica media) fra i punti del campione e la linea. Ora possiamo utilizzare il punto di intersezione e la pendenza che abbiamo appreso da questi dati per prevedere la variabile target per nuovi dati.

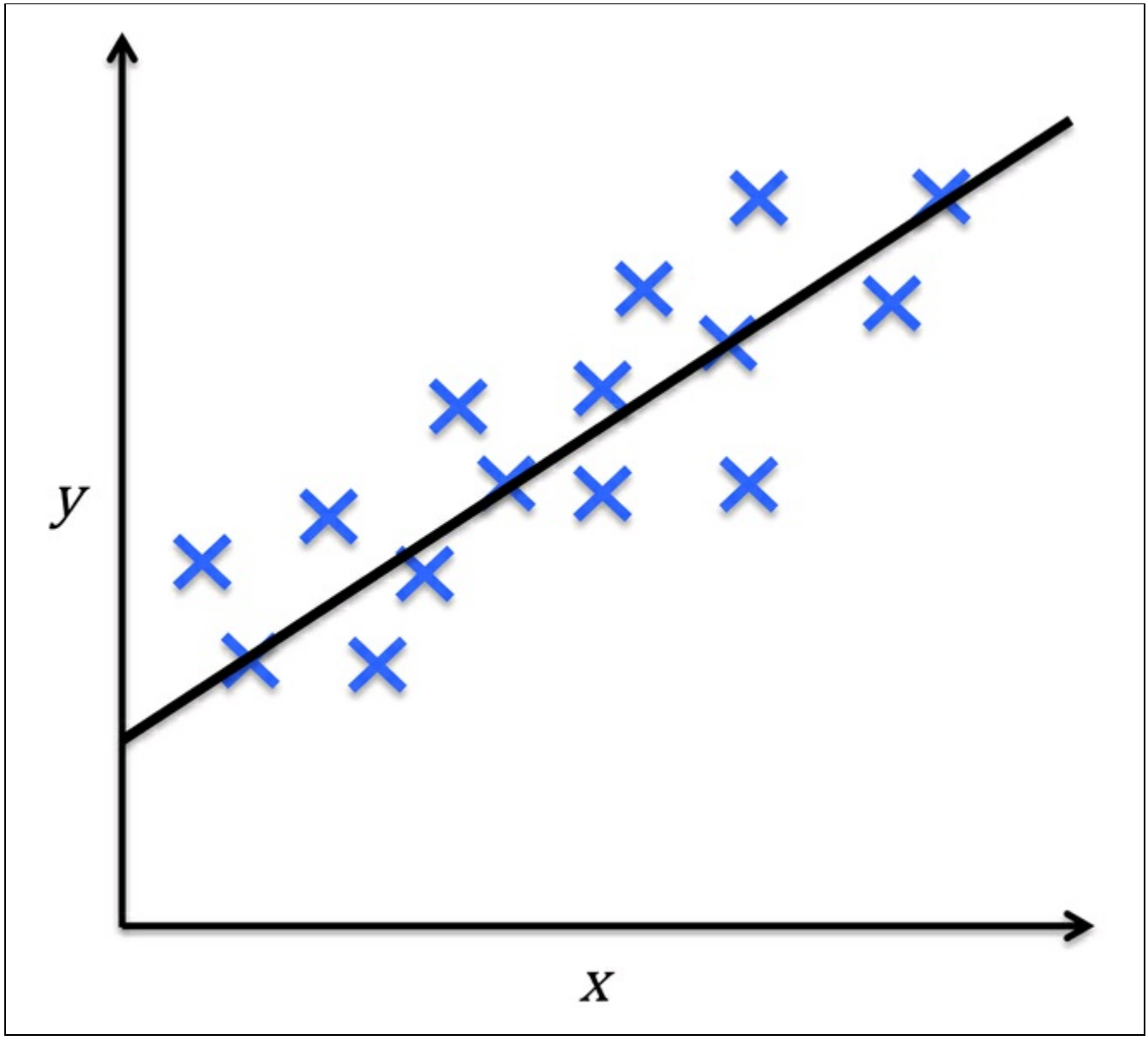


Figura 1.4

Risolvere problemi interattivi con l'apprendimento di rafforzamento

Un altro tipo di apprendimento automatico è l'apprendimento di rafforzamento. Qui l'obiettivo è quello di sviluppare un sistema (*agente*) che migliori le proprie

prestazioni sulla base delle interazioni con l'*ambiente*. Poiché, tipicamente, le informazioni relative allo stato corrente dell'ambiente includono anche un cosiddetto segnale di *ricompensa* (*reward*), possiamo considerare l'apprendimento di rafforzamento come un esempio di apprendimento *con supervisione*. Tuttavia, nell'apprendimento di rafforzamento, questo feedback non è l'etichetta o il valore corretto di verità, ma una misura della qualità con cui l'azione è stata misurata da una funzione di *ricompensa*. Tramite l'interazione con l'ambiente, un agente può quindi utilizzare l'apprendimento di rafforzamento per imparare una serie di *azioni* che massimizzano questa *ricompensa* tramite un approccio esplorativo del tipo trial-and-error o una pianificazione deliberativa.

Un esempio classico di apprendimento di rafforzamento è il motore del gioco degli scacchi. Qui, l'agente decide come svolgere una serie di mosse a seconda dello stato della scacchiera (l'ambiente) e la ricompensa può essere definita come la *vittoria* o la *sconfitta* alla fine del gioco (Figura 1.5).

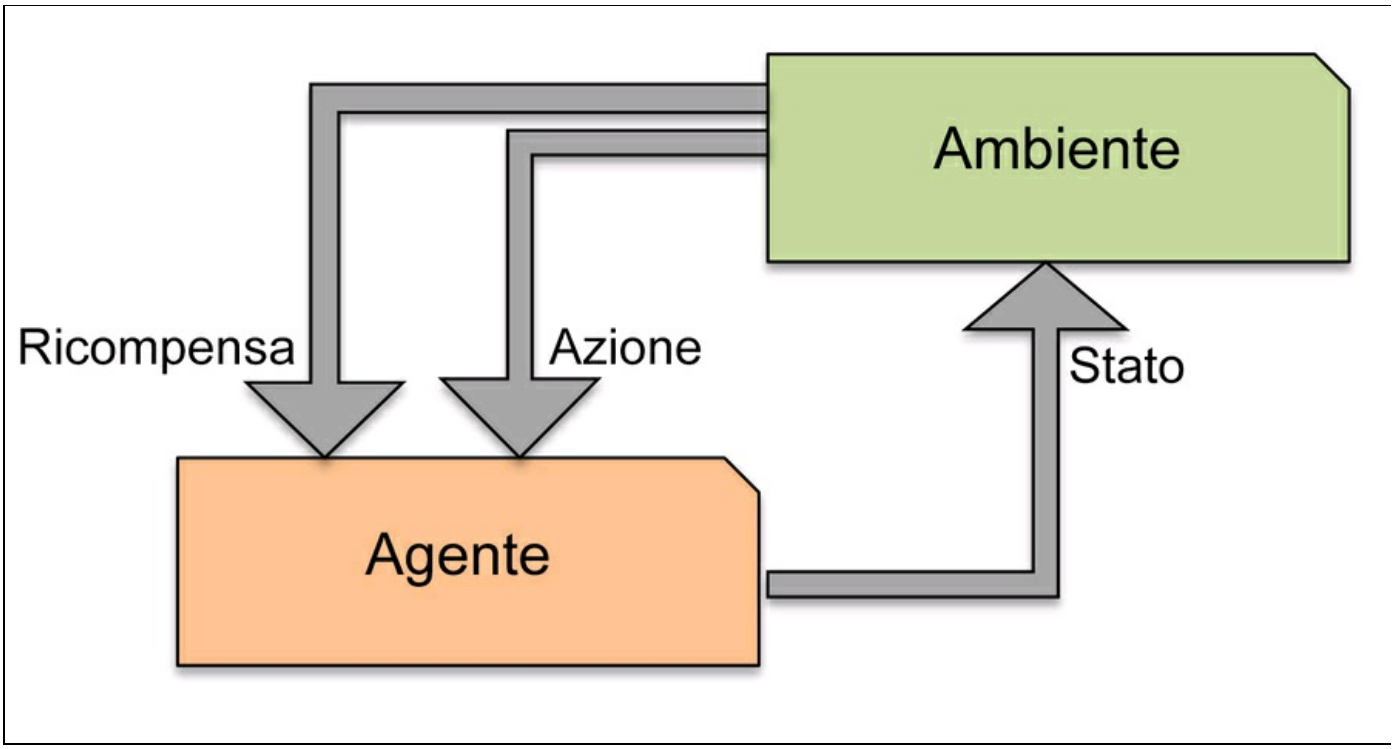


Figura 1.5

Scoprire le strutture nascoste con l'apprendimento senza supervisione

Nell'apprendimento con supervisione, conosciamo in anticipo la *risposta corretta* quando descriviamo il nostro modello, mentre nell'apprendimento di

rafforzamento definiamo una misura, o *ricompensa*, per le specifiche azioni messe in atto dall'agente. Nell'apprendimento senza supervisione, al contrario, abbiamo a che fare con dati non etichettati o dati dalla *struttura ignota*. Utilizzando tecniche di apprendimento senza supervisione, siamo in grado di osservare la struttura dei nostri dati, per estrarre da essi informazioni cariche di significato senza però poter contare sulla guida né di una variabile nota relativa al risultato, né una funzione di ricompensa.

Ricerca di sottogruppi tramite il clustering

Il *clustering* è una tecnica esplorativa di analisi dei dati che ci consente di organizzare una serie di informazioni all'interno di gruppi significativi (i *cluster*) senza avere alcuna precedente conoscenza delle appartenenze a tali gruppi. Ogni cluster che può essere derivato durante l'analisi definisce un gruppo di oggetti che condividono un certo grado di similarità, ma che sono più dissimili rispetto agli oggetti presenti negli altri cluster, motivo per cui il clustering viene talvolta chiamato "classificazione senza supervisione". Il clustering è un'ottima tecnica per la strutturazione dell'informazione e per individuare relazioni significative nei dati. Per esempio, consente agli operatori di marketing di individuare dei gruppi di clienti sulla base dei loro interessi, in modo da sviluppare specifici programmi di marketing.

La Figura 1.6 illustra il modo in cui il clustering può essere applicato all'organizzazione di alcuni dati senza etichetta suddividendoli in tre gruppi distinti sulla base della similarità delle caratteristiche x_1 e x_2.

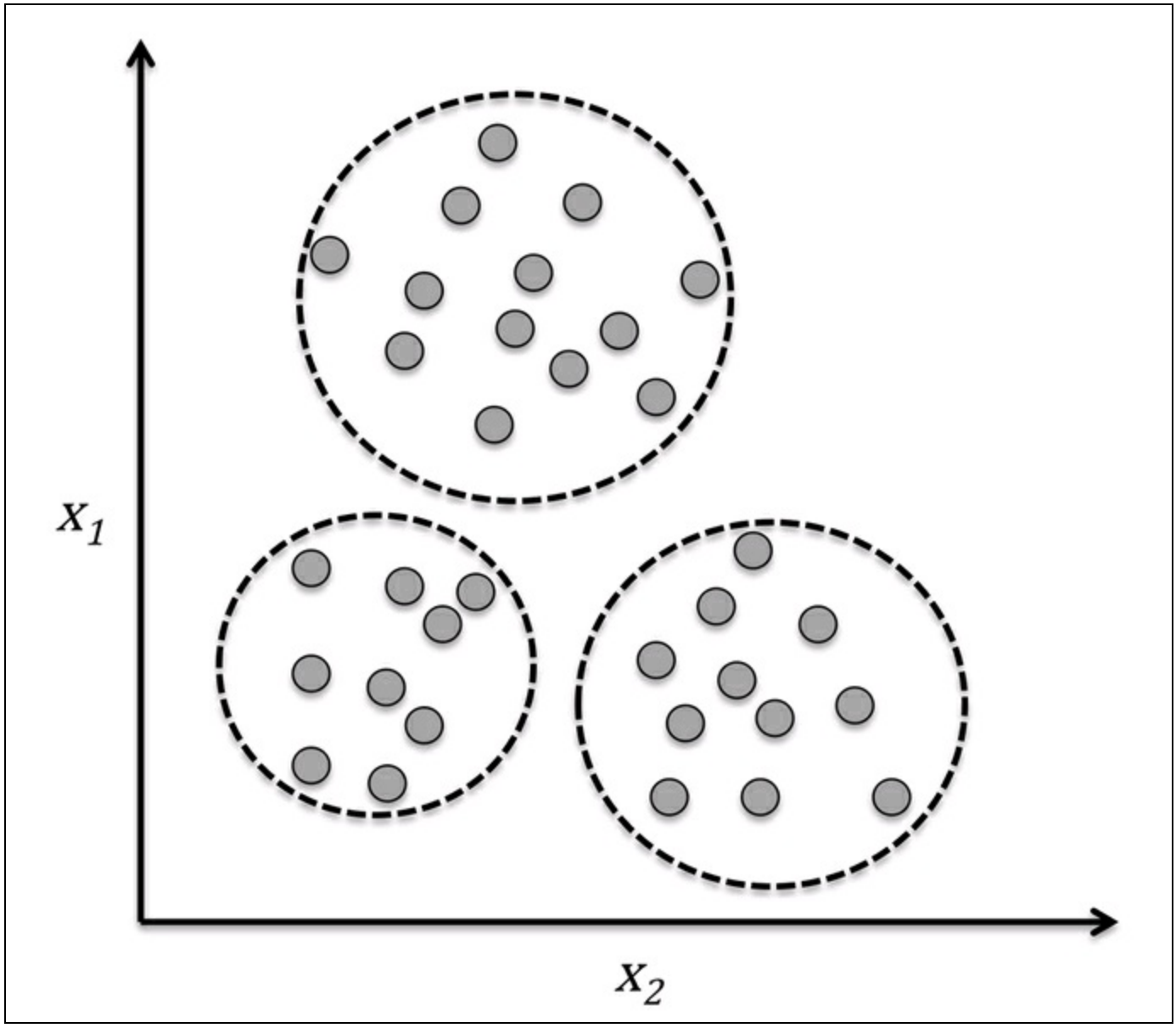


Figura 1.6

Riduzione dimensionale per la compressione dei dati

Un altro sottocampo dell'apprendimento senza supervisione è la *riduzione dimensionale*. Spesso ci troviamo a operare con dati a elevata dimensionalità (ogni osservazione fornisce un elevato numero di misure) il che può rappresentare una sfida in termini di spazio di memorizzazione disponibile e prestazioni computazionali degli algoritmi di apprendimento automatico. La riduzione dimensionale senza supervisione è un approccio comunemente utilizzato nella pre-elaborazione delle caratteristiche, e ha lo scopo di eliminare dai dati il "rumore", che può anche introdurre un degrado delle prestazioni predittive di alcuni algoritmi,

e di comprimere i dati in un sottospazio dimensionale più compatto, mantenendo però la maggior parte delle informazioni rilevanti.

Talvolta la riduzione dimensionale può essere utile anche per rappresentare i dati: per esempio, un insieme di caratteristiche a elevata dimensionalità può essere proiettato su uno spazio di caratteristiche mono-, bi- o tri-dimensionale, in modo da visualizzarlo tramite diagrammi o programmi 3D o 2D. La Figura 1.7 mostra un esempio in cui la riduzione della dimensionalità non lineare è stata applicata in modo da comprimere un grafico 3D *Swiss Roll* in un nuovo sottospazio bidimensionale delle caratteristiche.

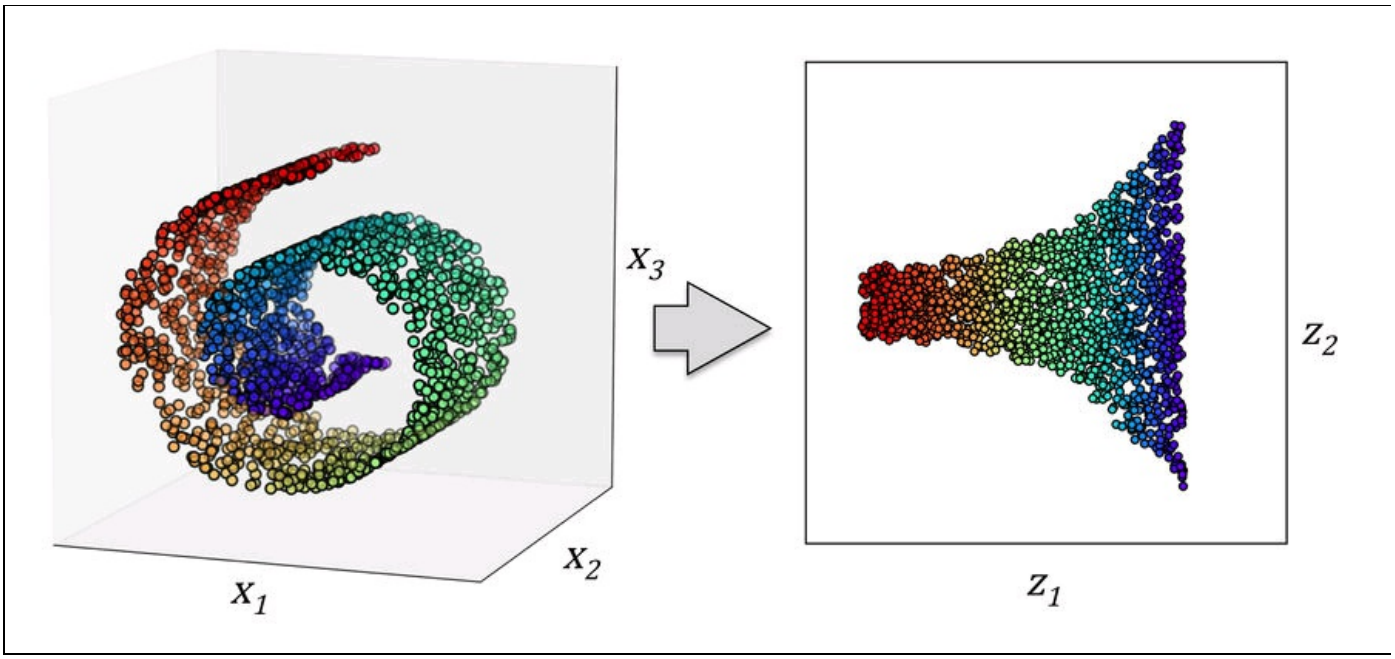
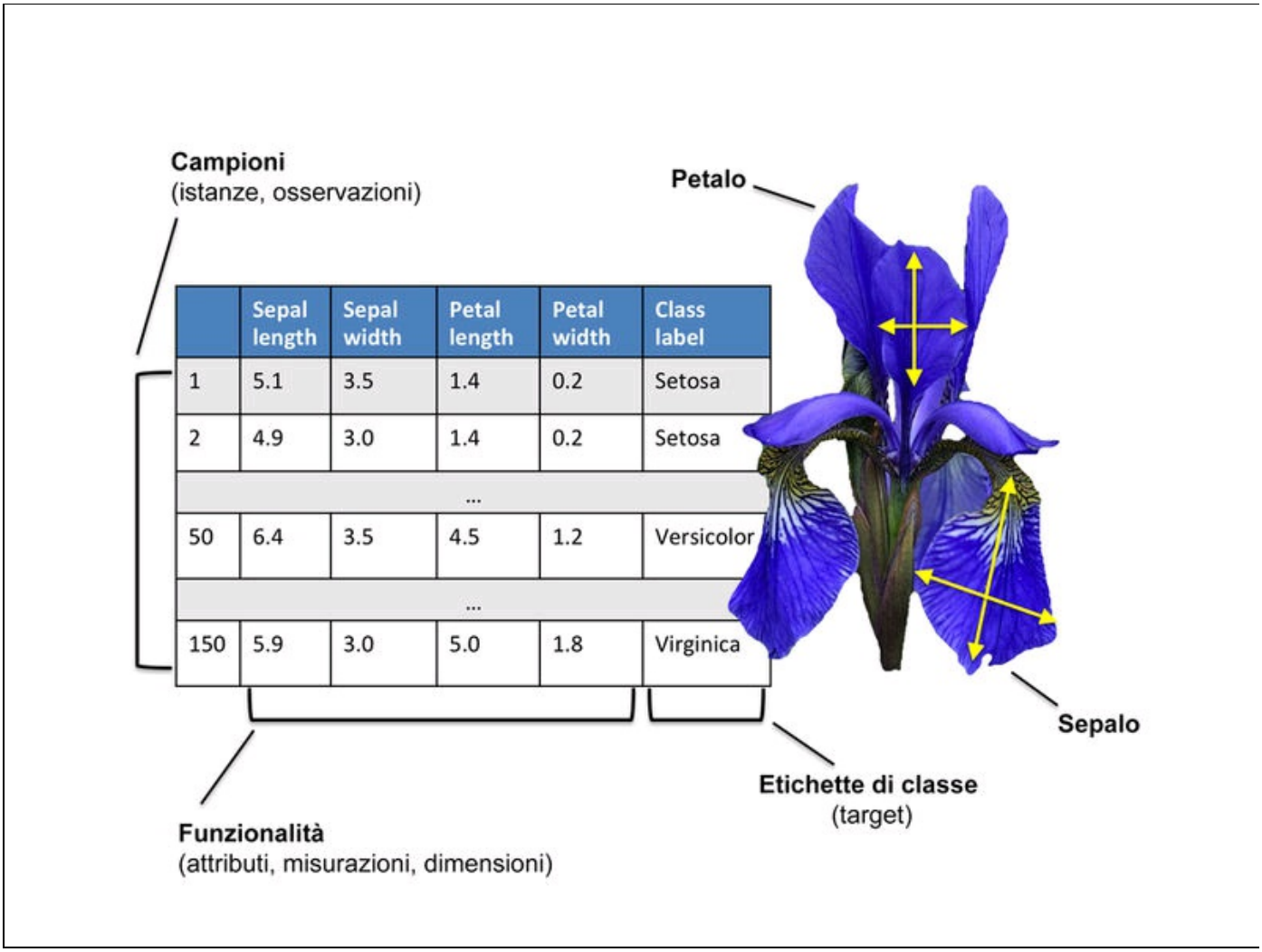


Figura 1.7

Introduzione alla terminologia e alla notazione di base

Ora che abbiamo introdotto le tre grandi categorie dell'apprendimento automatico (con supervisione, senza supervisione e di rafforzamento), diamo un'occhiata alla terminologia di base che ci troveremo a utilizzare nel corso dei prossimi capitoli. La seguente tabella rappresenta un estratto del dataset *Iris*, che è un classico esempio nel campo dell'apprendimento automatico. Il dataset Iris contiene la misurazione di centocinquanta specie di fiori iris di tre diverse specie: *Setosa*, *Versicolor* e *Virginica*. Qui, ogni campione di fiore rappresenta una riga del dataset e la misurazione del fiore in centimetri viene indicata in colonne, che rappresentano le caratteristiche del dataset (Figura 1.8).



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Figura 1.8

Per garantire la semplicità e l'efficienza della notazione e dell'implementazione, faremo uso di alcuni elementi di base dell'*algebra lineare*. Nei prossimi capitoli, utilizzeremo una notazione a *matrici* e *vettori* per far riferimento ai dati. Seguiremo la convenzione comune di rappresentare ciascun campione come una riga distinta nella matrice delle caratteristiche, *X*, dove ciascuna caratteristica è conservata come una colonna distinta.

Il dataset Iris, costituito da centocinquanta campioni e quattro caratteristiche, può pertanto essere scritto come una matrice 150×4 $\boldsymbol{X} \in \mathbb{R}^{150\times4}$:

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

Approfondimento

Per la parte rimanente del libro, utilizzeremo il numero in apice (*i*) per fare riferimento all'i-esimo campione di addestramento e il numero in pedice (*j*) per far riferimento alla j-esima dimensione del dataset di apprendimento.

Utilizzeremo lettere minuscole in grassetto per far riferimento ai vettori $\left(\boldsymbol{x} \in \mathbb{R}^{n\times1}\right)$ e lettere maiuscole in grassetto per far riferimento alle matrici $(\boldsymbol{X} \in \mathbb{R}^{n\times m})$. Per far riferimento ai singoli elementi di un vettore o di una matrice, utilizzeremo lettere in corsivo ($x^{(n)}$ o $x_{(m)}^{(n)}$).

Per esempio, x_1^{150} fa riferimento alla prima dimensione del centocinquantesimo campione di fiori, la *larghezza del sepalo*. Pertanto, ciascuna riga di questa matrice delle caratteristiche rappresenta l'istanza di un fiore e può essere scritta come un vettore a riga a 4 dimensioni $\boldsymbol{x}^{(i)} \in \mathbb{R}^{1\times4}$,

$$\boldsymbol{x}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$$

Ogni dimensione relativa a una caratteristica è invece un vettore a colonna a 150 dimensioni $\boldsymbol{x}^{(i)} \in \mathbb{R}^{150\times1}$, per esempio:

$$\boldsymbol{x}_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}$$

Analogamente, memorizzeremo le variabili target (in questo caso le etichette delle classi) come un vettore a colonna a 150 dimensioni

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} \left(y \in \{\text{Setosa, Versicolor, Virginica}\} \right)$$

Una roadmap per la realizzazione di sistemi di apprendimento automatico

Nei paragrafi precedenti abbiamo trattato i concetti di base dell'apprendimento automatico e i tre diversi tipi di apprendimento. In questo paragrafo ci concentreremo su altri elementi importanti di un sistema di apprendimento automatico che contraddistinguono l'algoritmo di apprendimento. Lo schema rappresentato nella Figura 1.9 rappresenta un tipico diagramma di flusso per l'impiego dell'apprendimento automatico nella *modellazione predittiva*, di cui parleremo nei prossimi paragrafi.

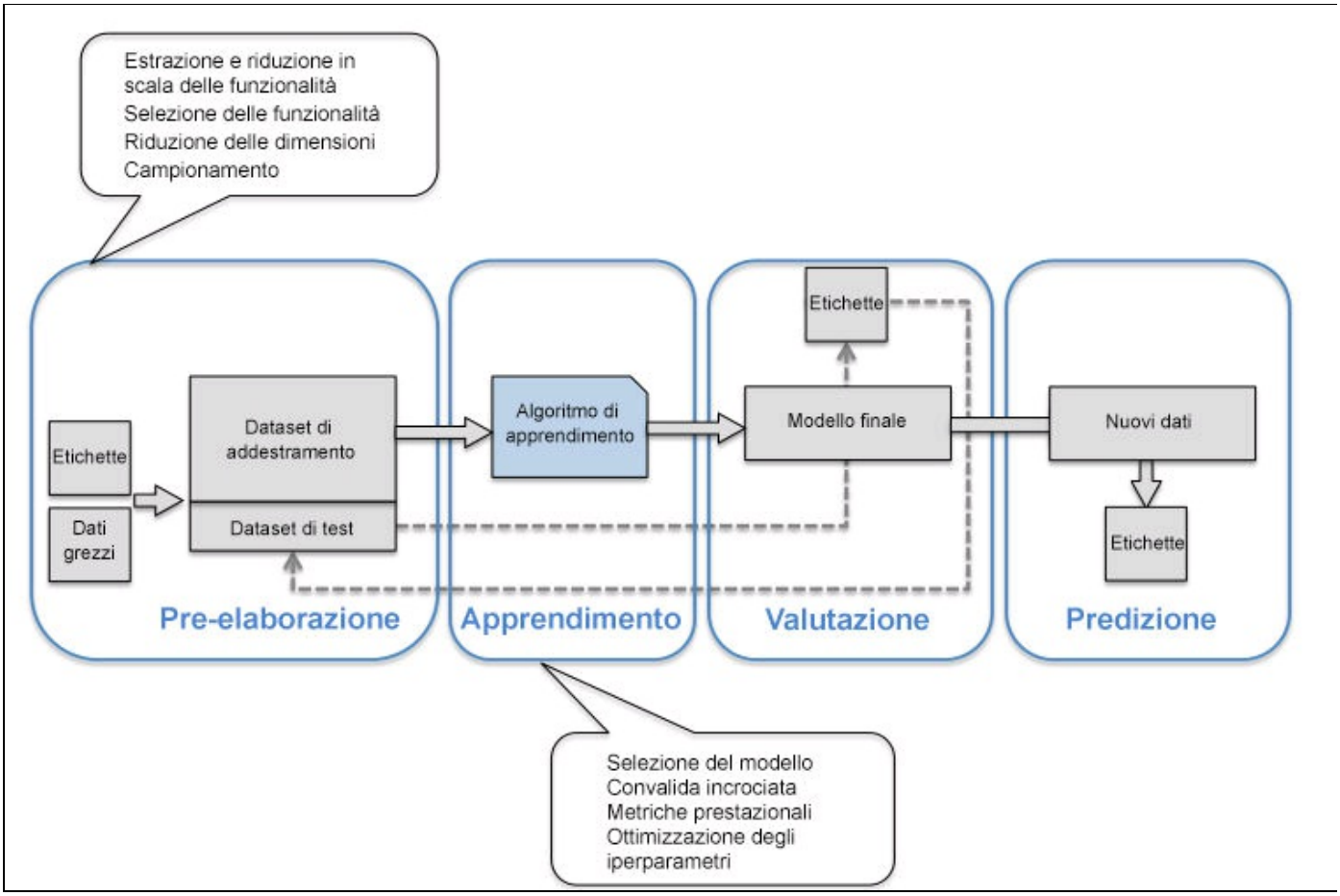


Figura 1.9

Pre-elaborazione: dare una "forma" ai dati

È raro che i dati abbiano una forma e un aspetto atti a garantire prestazioni ottimali dell'algoritmo di apprendimento. Pertanto, la *pre-elaborazione* dei dati è uno dei passi cruciali di qualsiasi applicazione di apprendimento automatico. Se

prendiamo ad esempio il dataset del fiore iris del paragrafo precedente, possiamo considerare i dati grezzi come una serie di fotografie del fiore, dalle quali possiamo estrarre caratteristiche significative. Le caratteristiche significative possono essere il colore, la tonalità, l'intensità dei toni, l'altezza, la larghezza e la profondità. Molti algoritmi di apprendimento automatico richiedono anche che, per ottenere le massime prestazioni, le caratteristiche scelte adottino la stessa scala, il che spesso viene ottenuto trasformando le caratteristiche in un intervallo [0, 1] oppure in una distribuzione normale standard con media 0 e varianza 1, come vedremo nei prossimi capitoli.

Alcune delle caratteristiche scelte possono avere un'elevata correlazione e, pertanto, risultare in qualche modo ridondanti. In tali casi, le tecniche di riduzione delle dimensioni sono utili per comprimere le caratteristiche in un sottospazio dimensionale inferiore. La riduzione della dimensionalità dello spazio delle caratteristiche presenta anche il vantaggio di richiedere meno spazio di memorizzazione e di accelerare il funzionamento dell'algoritmo di apprendimento.

Per determinare se il nostro algoritmo di apprendimento automatico non solo si comporta bene sul set di addestramento, ma esegue generalizzazioni corrette sui nuovi dati, potremmo voler suddividere in modo casuale il dataset in due set distinti: di addestramento e di test. Utilizziamo il set di addestramento per informare e ottimizzare il modello di apprendimento automatico, mentre teniamo da parte fino all'ultimo il set di test, per valutare il modello finale.

Addestramento e selezione di un modello predittivo

Come vedremo nei prossimi capitoli, sono stati sviluppati vari algoritmi di apprendimento automatico con lo scopo di risolvere problemi differenti. Un elemento importante che può essere tratto dal noto *No Free Lunch Theorems* di David Wolpert è che non esiste alcun apprendimento "gratuito" (*The Lack of A Priori Distinctions Between Learning Algorithms*, D.H. Wolpert 1996; *No Free Lunch Theorems for Optimization*, D.H. Wolpert e W.G. Macready, 1997). Intuitivamente, possiamo correlare questo concetto con il noto detto "Suppongo che sia allettante, quando l'unico strumento che hai a disposizione è un martello, trattare tutto come se fosse un chiodo" (Abraham Maslow, 1966). Per esempio, ogni algoritmo di classificazione ha i suoi difetti intrinseci e nessun modello di classificazione può vantare una superiorità assoluta se non abbiamo alcuna informazione sul compito da svolgere. In pratica, è pertanto essenziale confrontare

almeno un certo gruppo di algoritmi differenti, in modo da addestrarli e selezionare poi il modello che offre le migliori prestazioni. Ma prima di poter confrontare modelli differenti, dobbiamo decidere le metriche da impiegare per misurarne le prestazioni. Una metrica comunemente utilizzata è l'accuratezza della classificazione, che è definita come la proporzione tra le istanze classificate correttamente.

Sorge una domanda legittima: *come possiamo capire quale modello si comporta meglio sul dataset di test finale e sui dati reali se non utilizziamo questo dataset di test per la scelta del modello ma lo conserviamo per la valutazione finale del modello stesso?* Per poter risolvere il problema insito in questa domanda, possono essere utilizzate varie tecniche di convalida incrociata, nelle quali il dataset di addestramento viene ulteriormente suddiviso in *sottoinsiemi di addestramento di convalida*, in modo da stimare le *prestazioni di generalizzazione* del modello. Infine, non possiamo neppure aspettarci che i parametri standard dei vari algoritmi di apprendimento forniti dalle librerie di software siano ottimali per il nostro specifico problema. Pertanto, faremo frequentemente uso di *tecniche di ottimizzazione* degli iperparametri, che ci aiuteranno a ottimizzare le prestazioni del modello, come vedremo negli ultimi capitoli. Intuitivamente, possiamo considerare questi iperparametri come parametri che non vengono appresi dai dati, ma rappresentano le "manopole" del modello, sulle quali possiamo intervenire per migliorarne le prestazioni, come vedremo con maggiore chiarezza nei prossimi capitoli, quando le metteremo all'opera su esempi effettivi.

Valutazione dei modelli e previsione su istanze di dati mai viste prima

Dopo aver scelto un modello che possa adattarsi al dataset di addestramento, possiamo utilizzare il dataset di test per stimare la qualità della sua azione su dati mai visti prima, in modo da stimare l'errore di generalizzazione. Se siamo soddisfatti delle sue prestazioni, possiamo utilizzare questo modello anche per prevedere nuovi dati, futuri. È importante notare che i parametri delle procedure di cui abbiamo appena parlato (riduzione della scala e delle dimensioni delle caratteristiche) si possono ottenere solo dal dataset di addestramento e che questi stessi parametri vengono poi riapplicati per trasformare il dataset di test e anche ogni nuovo campione dei dati. Le prestazioni misurate sui dati di test, altrimenti, potrebbero essere eccessivamente ottimistiche.

Usare Python per attività di machine learning

Python è uno dei più noti linguaggi di programmazione per l'elaborazione dei dati e pertanto gode di una grande quantità di utili librerie aggiuntive, sviluppate dalla sua ottima comunità.

Sebbene le prestazioni dei linguaggi interpretati, come Python, per compiti intensivi dal punto di vista computazionale siano inferiori rispetto a quelle dei linguaggi di programmazione a basso livello, alcune librerie di estensione, come *NumPy* e *SciPy*, sono state sviluppate proprio basandosi su implementazioni a basso livello, in Fortran e C, in modo da accelerare le operazioni sui vettori, che devono operare su array multidimensionali.

Per attività di programmazione nel campo del machine learning, faremo riferimento principalmente alla libreria *scikit-learn*, che, attualmente, è una delle librerie open source di machine learning più note e diffuse.

Installazione dei pacchetti Python

Python è disponibile per i tre principali sistemi operativi, Microsoft Windows, Mac OS X e Linux, e il suo installer, come la documentazione, può essere scaricato dal sito web ufficiale di Python: https://www.python.org.

Questo libro è stato scritto per le versioni di Python successive alla 3.4.3 e si consiglia vivamente di scaricare sempre la versione più aggiornata di Python 3, sebbene la maggior parte degli esempi di codice possa essere compatibile anche con le versioni di Python successive alla 2.7.10. Se decidete di utilizzare Python 2.7 per eseguire gli esempi di codice, assicuratevi di considerare le grandi differenze che esistono fra le due versioni di Python. Un buon riepilogo delle differenze fra Python 3.4 e 2.7 si trova all'indirizzo https://wiki.python.org/moin/Python2orPython3.

Gli altri pacchetti che ci troveremo a utilizzare nel corso del libro possono essere installati tramite il programma installer *pip*, che fa parte della libreria standard di Python fin da Python 3.3. Ulteriori informazioni su pip si trovano all'indirizzo https://docs.python.org/3/installing/index.html.

Dopo aver scaricato e installato Python, possiamo eseguire pip dalla riga di comando, in modo da installare ulteriori pacchetti Python:

```
pip install SomePackage
```

I pacchetti già installati possono invece essere aggiornati tramite il flag --upgrade:

```
pip install SomePackage --upgrade
```

Una distribuzione Python alternativa, altamente consigliata per il calcolo scientifico, è Anaconda di Continuum Analytics. Anaconda è una distribuzione Python di livello professionale completamente gratuita (includendo anche gli utilizzi commerciali) che offre tutti i pacchetti essenziali di Python per attività scientifiche, matematiche e ingegneristiche, il tutto incluso in un'unica comoda distribuzione multipiattaforma. L'installer di Anaconda può essere scaricato all'indirizzo http://continuum.io/downloads#py34, mentre una guida rapida all'utilizzo di Anaconda è disponibile all'indirizzo https://store.continuum.io/static/img/Anaconda-Quickstart.pdf.

Dopo aver installato Anaconda, possiamo installare i nuovi pacchetti Python utilizzando il seguente comando:

```
conda install SomePackage
```

I pacchetti preesistenti possono essere aggiornati utilizzando il seguente comando:

```
conda update SomePackage
```

Nel corso del libro, utilizzeremo principalmente gli array multidimensionali *NumPy* per conservare e manipolare i dati. Occasionalmente, faremo uso di *pandas*, una libreria basata su NumPy che fornisce ulteriori strumenti per la manipolazione ad alto livello dei dati, che semplificano ancora di più l'elaborazione di dati di tipo tabulare. Per migliorare l'esperienza di apprendimento e consentire una visualizzazione dei dati quantitativi, aspetto che spesso è estremamente utile per comprendere, intuitivamente, il senso delle operazioni, utilizzeremo la libreria *matplotlib*, che presenta il vantaggio di essere molto personalizzabile.

I numeri di versione dei principali pacchetti Python che sono stati utilizzati per scrivere questo libro sono i seguenti. Assicuratevi che i numeri di versione dei pacchetti che avete installato siano uguali o successivi rispetto a questi numeri di versione, in modo da garantire che gli esempi di codice funzionino correttamente:

- NumPy 1.9.1
- SciPy 0.14.0
- scikit-learn 0.15.2
- matplotlib 1.4.0
- pandas 0.15.2

Riepilogo

In questo capitolo abbiamo esplorato le tecniche di machine learning (apprendimento automatico) da un punto di vista davvero molto elevato, con il solo scopo di familiarizzare con l'argomento e con i principali concetti che andremo a esplorare più in dettaglio nel corso dei prossimi capitoli.

Come abbiamo visto, l'apprendimento con supervisione è costituito da due importanti sottocampi: la classificazione e la regressione. Mentre i modelli di classificazione ci consentono di catalogare gli oggetti in classi note, possiamo utilizzare l'analisi di regressione per prevedere i risultati continui delle variabili target. L'apprendimento senza supervisione non solo ci offre tecniche utili per individuare le strutture all'interno di dati non etichettati, ma può essere utile anche per la compressione dei dati nei passi di pre-elaborazione delle caratteristiche.

Abbiamo proseguito con il tipico percorso di applicazione delle tecniche di machine learning alla soluzione dei problemi, che utilizzeremo come base per discussioni più approfondite e per esempi pratici, nel corso dei prossimi capitoli. Alla fine abbiamo configurato l'ambiente Python e abbiamo installato e aggiornato i pacchetti richiesti, in modo da prepararci a vedere in azione le attività di machine learning.

Nel prossimo capitolo, implementeremo uno dei primi algoritmi di classificazione ad apprendimento automatico, il che ci preparerà per il Capitolo 3, *I classificatori di machine learning di scikit-learn*, dove tratteremo algoritmi di apprendimento automatizzato più avanzati, che fanno uso della libreria open source di machine learning scikit-learn. Poiché gli algoritmi di machine learning apprendono dai dati, è fondamentale fornire loro informazioni utili e nel Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione*, ci occuperemo delle principali tecniche di pre-elaborazione. Nel Capitolo 5, *Compressione dei dati tramite la riduzione della dimensionalità*, affronteremo l'argomento delle tecniche di riduzione della dimensionalità, che possono aiutarci a comprimere il dataset in un sottospazio delle caratteristiche dotato di un numero inferiore di dimensioni, il che può essere utile per migliorarne l'efficienza computazionale. Un aspetto importante della costruzione di modelli di machine learning consiste nella valutazione delle loro prestazioni e nella stima della qualità delle loro previsioni su dati nuovi, imprevisti. Nel Capitolo 6, *Valutazione dei modelli e ottimizzazione degli iperparametri*, apprenderemo tutte le migliori tecniche di ottimizzazione e

valutazione dei modelli. In alcune situazioni potremmo non essere ancora soddisfatti delle prestazioni del nostro modello predittivo, sebbene abbiamo dedicato ore o magari giorni a ottimizzarlo e collaudarlo. Nel Capitolo 7, *Combinare più modelli: l'apprendimento d'insieme*, impareremo a combinare più modelli di machine learning, per realizzare sistemi predittivi ancora più potenti.

Dopo aver trattato tutti i concetti più importanti di una tipica catena di machine learning, implementeremo un modello per la previsione delle emozioni nel testo nel Capitolo 8, *Tecniche di machine learning per l'analisi del sentiment*, e poi, nel Capitolo 9, *Embedding di un modello in un'applicazione web*, includeremo il tutto in un'applicazione web aperta ai visitatori. Nel Capitolo 10, *Previsioni di variabili target continue: l'analisi a regressione* utilizzeremo gli algoritmi di machine learning per l'analisi della regressione, che ci consentirà di prevedere delle variabili di output continue, infine nel Capitolo 11, *Lavorare con dati senza etichette: l'analisi a cluster*, applicheremo gli algoritmi di clustering, che ci consentiranno di trovare quelle strutture che si nascondono all'interno dei dati. L'ultimo capitolo di questo libro affronterà le reti neurali artificiali, che ci consentiranno di affrontare problemi complessi, come il riconoscimento delle immagini e del parlato, uno degli argomenti attualmente più interessanti nella ricerca nell'ambito del machine learning.

Addestrare gli algoritmi a compiti di classificazione

In questo capitolo utilizzeremo i primi algoritmi di machine learning storicamente descritti per la classificazione, il *perceptron* e i neuroni adattativi lineari (*adaptive linear neurons*). Inizieremo implementando un perceptron passo dopo passo in Python e addestrandolo a classificare le diverse specie di fiori del dataset Iris. Questo ci aiuterà a comprendere il funzionamento degli algoritmi di machine learning per la classificazione e come essi possono essere implementati efficacemente in Python. Trattando le basi dell'ottimizzazione tramite i neuroni lineari adattativi, prepareremo il campo per l'utilizzo di classificatori più potenti tramite la libreria di machine learning scikit-learn, di cui parleremo nel Capitolo 3, *I classificatori di machine learning di scikit-learn*.

Gli argomenti che tratteremo in questo capitolo sono i seguenti:

- le basi della realizzazione di algoritmi di machine learning;
- uso di pandas, NumPy e matplotlib per leggere, elaborare e rappresentare i dati;
- implementazione di algoritmi di classificazione lineare in Python.

Neuroni artificiali: breve introduzione ai primordi del machine learning

Prima di parlare più in dettaglio del perceptron e di altri algoritmi correlati, è il caso di trattare brevemente i primordi dell'apprendimento automatico. Cercando di comprendere come funzionasse il cervello biologico per sfruttarne i principi nell'intelligenza artificiale, Warren McCullock e Walter Pitts hanno pubblicato nel 1943 un primo schema di cellula semplificata del cervello, il cosiddetto *neurone di McCullock-Pitts* (*MCP*) (W. S. McCulloch e W. Pitts, *A Logical Calculus of the Ideas Immanent* in "Nervous Activity. The bulletin of mathematical biophysics", 5(4):115–133, 1943). I neuroni sono cellule nervose interconnesse che si trovano nel cervello e sono coinvolte nell'elaborazione e trasmissione di segnali chimici ed elettrici, come illustrato nella Figura 2.1.

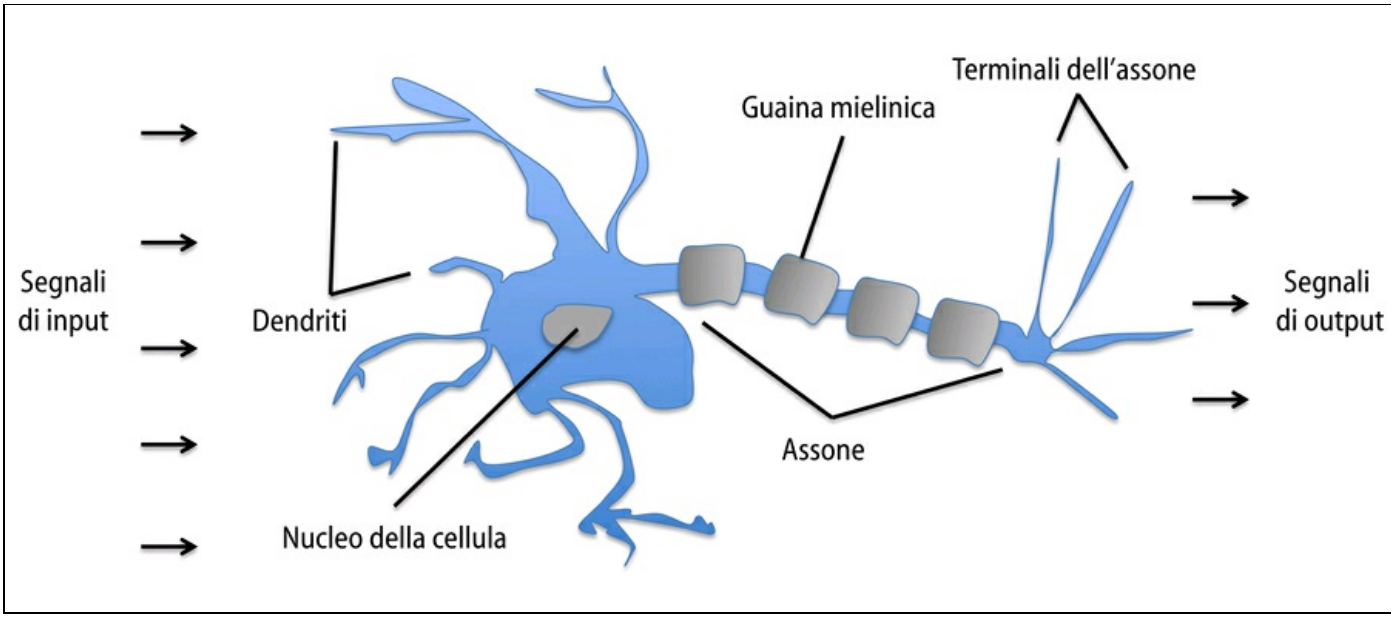


Figura 2.1

McCullock e Pitts hanno ricreato una cellula nervosa come un semplice gate logico con un output binario; ai dendriti arrivano più segnali, i quali entrano poi nel corpo della cellula; se il segnale accumulato supera una determinata soglia, viene prodotto un segnale di output, che viene poi passato tramite l'assone.

Solo pochi anni dopo, Frank Rosenblatt ha pubblicato il primo concetto della regola di apprendimento del perceptron basata sul modello del neurone MCP (F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton*, Cornell Aeronautical Laboratory, 1957). Con la sua regola di apprendimento perceptron,

Rosenblatt ha proposto un algoritmo che avrebbe appreso automaticamente i coefficienti di peso ottimali da moltiplicare con le caratteristiche di input in modo da poter prendere la decisione sul fatto che un neurone si attivi o meno. Nel contesto dell'apprendimento e della classificazione con supervisione, l'algoritmo potrebbe essere utilizzato per prevedere se un determinato campione appartiene a una classe o a un'altra.

In termini più formali, possiamo considerare questo problema come un compito di classificazione binaria, dove, per semplicità, facciamo riferimento alle nostre due classi come 1 (classe positiva) e -1 (classe negativa). Possiamo poi definire una *funzione di attivazione* $\phi(z)$ che prende una combinazione lineare di determinati valori di input $\boldsymbol{x}$ e un corrispondente vettore di pesi $\boldsymbol{w}$, dove z è il cosiddetto input della rete ($z = w_1 x_1 + \ldots + w_m x_m$):

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Ora, se l'attivazione di un particolare campione $\boldsymbol{x}^{(i)}$, ovvero l'output di $\phi(z)$, è maggiore di una determinata soglia θ, siamo in grado di prevedere che rientri nella classe 1, altrimenti è nella classe -1. Nell'algoritmo perceptron, la funzione di attivazione $\phi(\cdot)$ è una *funzione definita a tratti* (*piecewise-defined function*), che viene talvolta chiamata *funzione di passo Heaviside*:

$$\phi(z) = \begin{cases} 1 & if\ z \geq \theta \\ -1 & altrimenti \end{cases}$$

Per semplicità, possiamo portare la soglia θ sul lato sinistro dell'equazione e definire un peso zero come $w_0 = -\theta$ e $x_0 = 1$, in modo da poter scrivere $\mathbf{z}$ in una forma più compatta:

$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \boldsymbol{w}^T \boldsymbol{x}$$

e

$$\phi(z) = \begin{cases} 1 & if\ z \geq 0 \\ -1 & altrimenti \end{cases}.$$

Approfondimento

Nei prossimi paragrafi, faremo frequentemente uso della notazione di base dell'algebra lineare. Per esempio, abbrevieremo la somma dei prodotti dei valori contenuti in $\boldsymbol{x}$ e $\boldsymbol{w}$ utilizzando un prodotto vettoriale, mentre la lettera T all'apice sta per *trasposizione*, un'operazione che trasforma un vettore colonna in un vettore riga e viceversa:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} \boldsymbol{x}_j \boldsymbol{w}_j = \boldsymbol{w}^T \boldsymbol{x}$$

per esempio:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$$

L'operazione di trasposizione può essere applicata anche a una matrice, per rifletterla rispetto alla diagonale, come nel seguente esempio:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

In questo libro, utilizzeremo solo i concetti più semplici dell'algebra lineare. Tuttavia, chi avesse bisogno di un breve ripasso, può dare un'occhiata al testo *Linear Algebra Review and Reference* di Zico Kolter, disponibile gratuitamente all'indirizzo http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf.

La Figura 2.2 illustra come l'input della rete $z = \boldsymbol{w}^T \boldsymbol{x}$ viene ridotto in un output binario (-1 o 1) dalla funzione di attivazione del perceptron (lato sinistro della figura) e come possa essere utilizzato per discriminare fra due classi separabili in modo lineare (lato destro della figura).

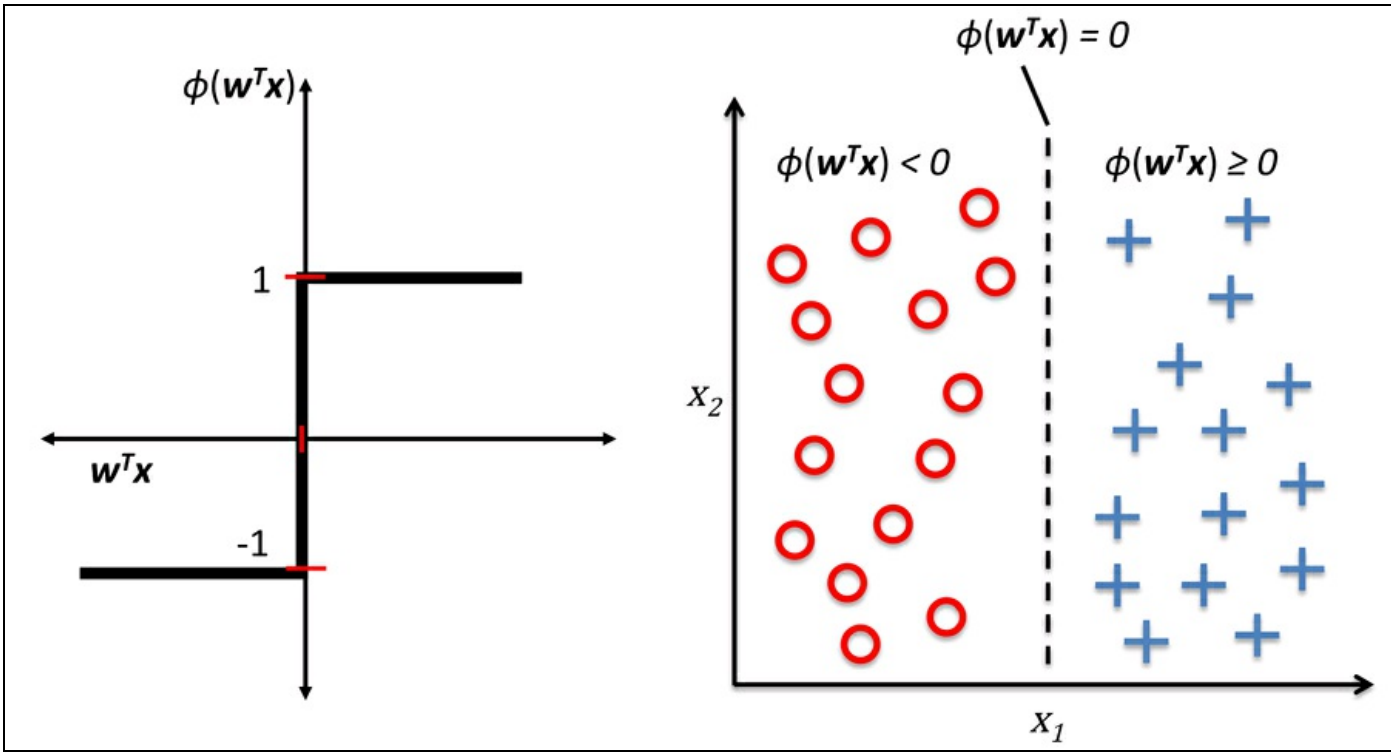


Figura 2.2

L'idea su cui si basa il neurone MCP e il modello di perceptron di Rosenblatt a soglia è il fatto di utilizzare un approccio riduzionista per emulare il modo in cui funziona un singolo neurone del cervello, il quale può *attivarsi* oppure non attivarsi. Pertanto, la regola del perceptron iniziale di Rosenblatt è piuttosto semplice e può essere riassunta dai seguenti passi.

1. Inizializzare i pesi a 0 o a numeri casuali piccoli.
2. Per ogni campione di addestramento $\boldsymbol{x}^{(i)}$ svolgere seguenti passi.
    1. Calcolare il valore di output $\hat{y}$.
    2. Aggiornare i pesi.

Qui, il valore di output è l'etichetta della classe prevista dalla funzione a passo unitario che abbiamo definito in precedenza, e il simultaneo aggiornamento di ogni peso w_j nel vettore dei pesi $\boldsymbol{w}$ può essere scritto, più formalmente, come:

$$w_j := w_j + \Delta w_j$$

Il valore di Δw_j, che viene utilizzato per aggiornare il peso w_j, viene calcolato dalla regola di apprendimento del perceptron:

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$

dove η è il tasso di apprendimento (un valore costante compreso fra 0.0 e 1.0), $y^{(i)}$ è la vera etichetta della classe del campione di apprendimento i-esimo e $\hat{y}^{(i)}$ è l'etichetta della classe prevista. È importante notare che tutti i pesi del vettore dei pesi devono essere aggiornati simultaneamente, il che significa che non ricalcoleremo $\hat{y}^{(i)}$ prima che siano stati aggiornati tutti i pesi Δw_j. In concreto, per un dataset bidimensionale, scriveremo l'aggiornamento nel seguente modo:

$$\Delta w_0 = \eta\left(y^{(i)} - output^{(i)}\right)$$

$$\Delta w_1 = \eta\left(y^{(i)} - output^{(i)}\right)x_1^{(i)}$$

$$\Delta w_2 = \eta\left(y^{(i)} - output^{(i)}\right)x_2^{(i)}$$

Prima di implementare la regola del perceptron in Python, eseguiamo un semplice esperimento per illustrare quanto sia meravigliosamente semplice questa regola di apprendimento. Nelle due situazioni in cui il perceptron predice correttamente l'etichetta della classe, i pesi non vengono modificati:

$$\Delta w_j = \eta\left(-1--1\right)x_j^{(i)} = 0$$

$$\Delta w_j = \eta\left(1-1\right)x_j^{(i)} = 0$$

Tuttavia, nel caso di una predizione errata, i pesi vengono modificati verso la direzione della classe target positiva o negativa:

$$\Delta w_j = \eta\left(1--1\right)x_j^{(i)} = \eta\left(2\right)x_j^{(i)}$$

$$\Delta w_j = \eta\left(-1-1\right)x_j^{(i)} = \eta\left(-2\right)x_j^{(i)}$$

Per rendere più intuitivo l'uso del fattore moltiplicativo $x_j^{(i)}$, esaminiamo un altro semplice esempio, dove:

$$\hat{y}_j^{(i)} = +1,\ y^{(i)} = -1,\ \eta = 1$$

Supponiamo che $x_j^{(i)} = 0.5$ e di aver erroneamente classificato questo campione come -1. In questo caso, aumenteremo il peso corrispondente di 1, in modo che l'attivazione $x_j^{(i)} = w_j^{(i)}$ sia più positiva la prossima volta che incontreremo questo campione e pertanto sia più probabile che superi la soglia della funzione di passo unitario che consenta di classificare il campione come +1:

$$\Delta w_j^{(i)} = \left(1^{(i)} - -1^{(i)}\right) 0.5^{(i)} = (2) 0.5^{(i)} = 1$$

L'aggiornamento del peso è proporzionale al valore di $x_j^{(i)}$. Per esempio, se abbiamo un altro campione $x_j^{(i)} = 2$ che è stato erroneamente classificato come -1, spingeremo il confine decisionale di una quantità ancora più ampia, in modo che la prossima volta il campione venga classificato correttamente:

$$\Delta w_j = \left(1^{(i)} - -1^{(i)}\right) 2^{(i)} = (2) 2^{(i)} = 4$$

È importante notare che la convergenza del perceptron è garantita solo se le due classi sono separabili linearmente e se il tasso di apprendimento è sufficientemente ridotto. Se le due classi non possono essere separate da un confine decisionale lineare, possiamo definire un numero massimo di passi sul dataset di apprendimento (*epoch*) e/o una soglia per il numero di errate classificazioni che possiamo tollerare. In caso contrario, il perceptron non smetterebbe mai di correggere i pesi (Figura 2.3).

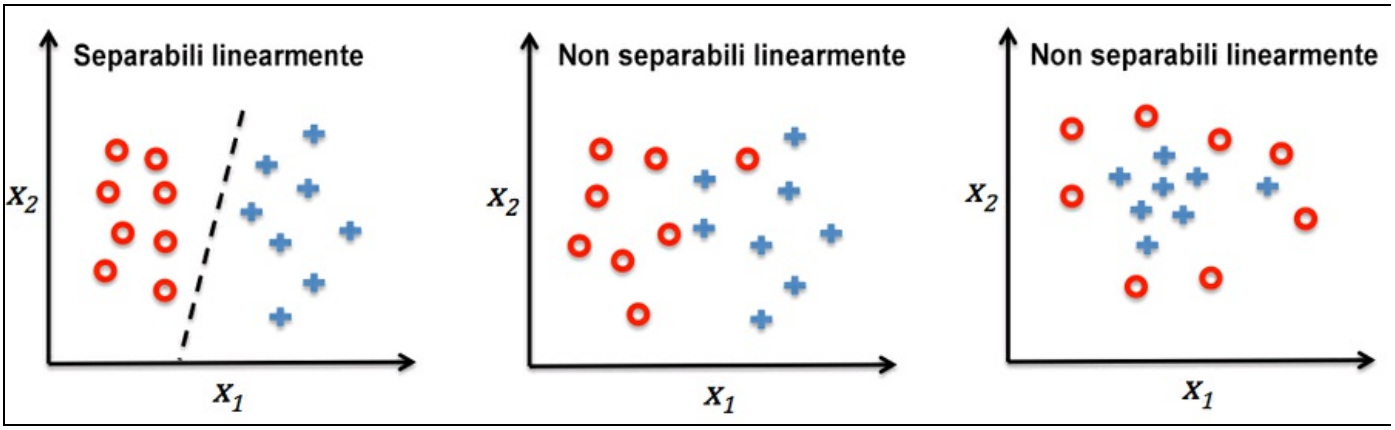


Figura 2.3

Ora, prima di passare all'implementazione, nel prossimo paragrafo, riepiloghiamo ciò che abbiamo appena appreso tramite una semplice figura (Figura 2.4) che illustra il concetto generale di funzionamento del perceptron.

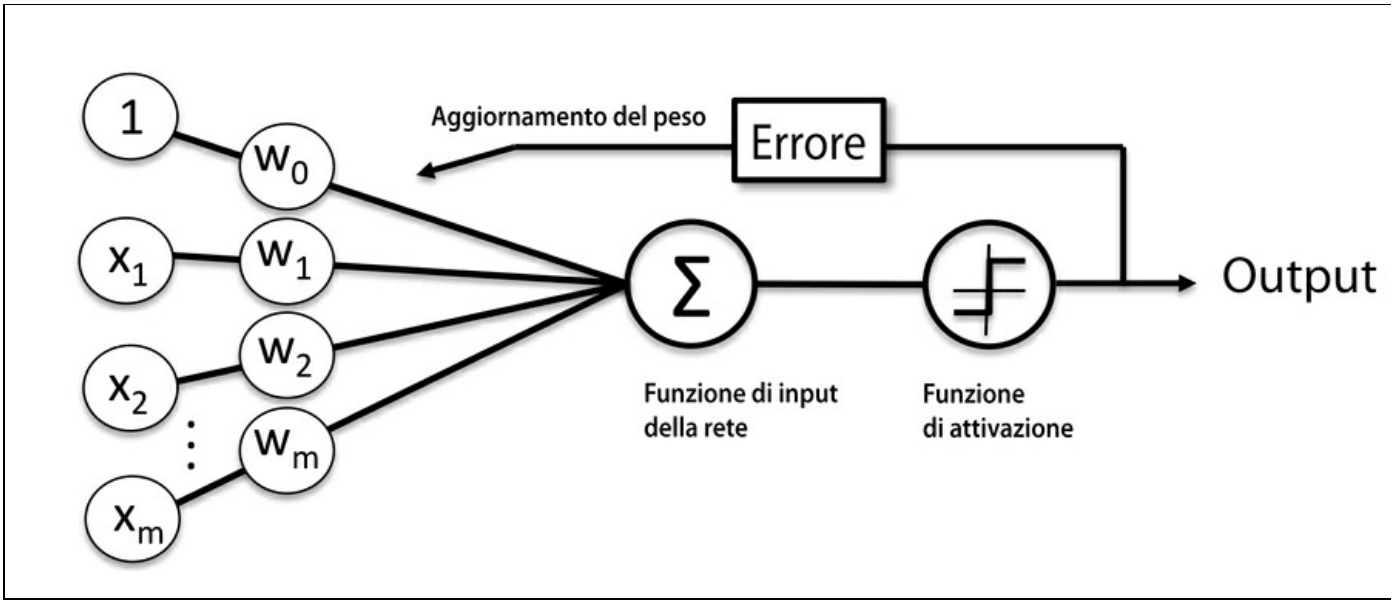


Figura 2.4

La figura precedente illustra il modo in cui il perceptron riceve gli input di un campione x e li combina con i pesi w per calcolare l'input della rete. L'input della rete viene poi passato alla funzione di attivazione (qui si tratta della funzione di passo unitario), che genera un output binario -1 o +1: l'etichetta della classe prevista per il campione. Durante la fase di apprendimento, questo output viene utilizzato per calcolare l'errore della previsione e aggiornare i pesi.

Implementazione in Python di un algoritmo di apprendimento perceptron

Nel paragrafo precedente abbiamo esaminato il funzionamento della regola del perceptron di Rosenblatt; procediamo e implementiamo tale regola in Python, applicandola poi al dataset Iris che abbiamo introdotto nel Capitolo 1, *Dare ai computer la capacità di apprendere dai dati*. Adotteremo un approccio a oggetti per definire l'interfaccia del perceptron come una `Class` Python, che ci consente di inizializzare i nuovi oggetti perceptron che possono imparare dai dati tramite un metodo `fit` ed effettuare previsioni tramite un metodo distinto `predict`. Per convenzione, aggiungeremo un carattere di sottolineatura agli attributi che non vengono creati dall'inizializzazione dell'oggetto, ma richiamando gli altri metodi dell'oggetto, per esempio, `self.w_`.

NOTA

Se non conoscete l'uso delle librerie scientifiche di Python o per un breve ripasso, potete contare sulle seguenti risorse.

NumPy: http://wiki.scipy.org/Tentative_NumPy_Tutorial

Pandas: http://pandas.pydata.org/pandas-docs/stable/tutorials.html

Matplotlib: http://matplotlib.org/users/beginner.html

Inoltre, per seguire meglio gli esempi di codice, scaricate i notebook IPython dal sito web Packt. Per un'introduzione generale ai notebook IPython, potete visitare l'indirizzo https://ipython.org/ipython-doc/3/notebook/index.html.

```
import numpy as np
class Perceptron(object):
    """Perceptron classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
      Weights after fitting.
    errors_ : list
      Number of misclassifications in every epoch.

    """

    def __init__(self, eta=0.01, n_iter=10):
      self.eta = eta
      self.n_iter = n_iter

    def fit(self, X, y):
      """Fit training data.
```

```
    Parameters
    ----------
    X : {array-like}, shape = [n_samples, n_features]
      Training vectors, where n_samples
      is the number of samples and
      n_features is the number of features.
    y : array-like, shape = [n_samples]
      Target values.

    Returns
    -------
    self : object

    """
    self.w_ = np.zeros(1 + X.shape[1])
    self.errors_ = []
    for _ in range(self.n_iter):
      errors = 0
      for xi, target in zip(X, y):
        update = self.eta * (target - self.predict(xi))
        self.w_[1:] += update * xi
        self.w_[0] += update
        errors += int(update != 0.0)
      self.errors_.append(errors)
    return self

  def net_input(self, X):
    """Calculate net input"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

  def predict(self, X):
    """Return class label after unit step"""
    return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Utilizzando questa implementazione del perceptron, possiamo inizializzare nuovi oggetti `Perceptron` con un determinato tasso di apprendimento `eta` e `n_iter`, che è il numero di epoch (passi all'interno del dataset di addestramento). Tramite il metodo `fit` inizializziamo i pesi in `self.w_` con un vettore zero $\mathbb{R}^{m+1}$, dove m è il numero di dimensioni (caratteristiche) del dataset dove aggiungiamo `1` per il peso zero (ovvero, la soglia).

NOTA

L'indicizzazione di NumPy per gli array monodimensionali opera in modo simile alle liste di Python utilizzando la notazione a parentesi quadre (`[]`). Per gli array bidimensionali, il primo indice fa riferimento al numero della riga e il secondo al numero della colonna. Per esempio, utilizziamo `X[2, 3]` per selezionare la terza riga e la quarta colonna di un array bidimensionale `X`.

Dopo che i pesi sono stati inizializzati, il metodo `fit` esegue un ciclo su tutti i singoli campioni del dataset di addestramento e aggiorna i pesi sulla base della regola di apprendimento del perceptron che abbiamo trattato nel paragrafo precedente. Le etichette delle classi vengono determinate sulla base del metodo `predict`, che viene utilizzato anche nel metodo `fit` per prevedere l'etichetta della classe per l'aggiornamento del peso, ma `predict` può essere utilizzato anche per prevedere le etichette delle classi dei nuovi dati dopo che abbiamo affinato il nostro modello. Inoltre, raccogliamo anche il numero di errate classificazioni durante ciascuna epoch nella lista `self.errors_`, in modo che, successivamente, potremo analizzare la

qualità di funzionamento del perceptron durante l'addestramento. La funzione `np.dot` che viene utilizzata nel metodo `net_input` non fa altro che calcolare il prodotto di vettori $\boldsymbol{w}^T\boldsymbol{x}$.

NOTA

Invece di utilizzare NumPy per calcolare il prodotto vettoriale fra i due array a e b tramite `a.dot(b)` o `np.dot(a, b)`, potremmo anche eseguire il calcolo direttamente in Python, utilizzando `sum([i*j for i,j in zip(a, b)]`. Tuttavia, il vantaggio di utilizzare NumPy rispetto alle classiche strutture a ciclo di Python è il fatto che le sue operazioni aritmetiche sono vettorializzate. Ciò significa che a tutti gli elementi dell'array viene automaticamente applicata un'operazione aritmetica elementare. Formulando le nostre operazioni aritmetiche come una sequenza di istruzioni di un array invece che svolgere un insieme di operazioni su ciascun elemento, uno alla volta, sfrutteremo al meglio le più recenti architetture di CPU, dotate di supporto della funzionalità *Single Instruction, Multiple Data* (SIMD). Inoltre, NumPy utilizza librerie di algebra lineare estremamente ottimizzate, come *Basic Linear Algebra Subprograms* (BLAS) e *Linear Algebra Package* (LAPACK) scritte in C o Fortran. Infine, NumPy ci consente di scrivere il nostro codice in una forma più compatta e intuitiva, utilizzando le basi dell'algebra lineare, come i prodotti fra vettori e matrici.

Addestrare un modello del perceptron sul dataset Iris

Per addestrare la nostra implementazione del perceptron, caricheremo dal dataset Iris le due classi di fiori *Setosa* e *Versicolor*. Sebbene la regola del perceptron non si limiti a due dimensioni, considereremo solo, per comodità di visualizzazione, le due caratteristiche *sepal length* (lunghezza del sepalo) e *petal length* (lunghezza del petalo). Inoltre, per motivi pratici, abbiamo scelto solo le due classi di fiori *Setosa* e *Versicolor*. Tuttavia, l'algoritmo del perceptron può essere esteso a una classificazione multiclasse, per esempio attraverso la tecnica *One-vs.-All*.

NOTA

One-vs.-All (OvA) chiamata nella letteratura anche *One-vs.-Rest (OvR)* è una tecnica utilizzata per estendere un classificatore binario a problemi che prevedono più classi. Utilizzando la tecnica OvA, possiamo addestrare un classificatore per classe, dove quella specifica classe viene trattata come la classe positiva e i campioni relativi a tutte le altre classi vengono considerati come appartenenti alla classe negativa. Per classificare un nuovo campione di dati, utilizzeremo *N* classificatori, dove *N* è il numero di etichette delle classi, assegnando l'etichetta della classe con l'affidabilità più elevata per quello specifico campione. Nel caso del perceptron, possiamo utilizzare la tecnica OvA per scegliere l'etichetta della classe che è dotata del valore di input della rete maggiore in termini assoluti.

Innanzitutto utilizziamo la libreria *pandas* per caricare il dataset Iris direttamente dall'*UCI Machine Learning Repository* all'oggetto `DataFrame` e stampiamo le ultime

cinque righe tramite il metodo `tail`, per verificare che i dati siano stati caricati correttamente:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
... 'machine-learning-databases/iris/iris.data', header=None)
>>> df.tail()
```

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Figura 2.5

Poi, estraiamo le prime 100 etichette delle classi che corrispondono a 50 fiori *Iris-Setosa* e 50 fiori *Iris-Versicolor*, convertendo le etichette delle classi nelle due etichette delle classi intere `1` (*Versicolor*) e `-1` (*Setosa*) che assegneremo a un vettore `y` dove il metodo dei valori di un `pandas DataFrame` fornisce la corrispondente rappresentazione NumPy. Analogamente, estraiamo la prima colonna di caratteristiche (*sepal length*) e la terza colonna di caratteristiche (*petal length*) di questi 100 campioni di addestramento e assegnamo tali valori a una matrice delle caratteristiche `X`, che possiamo rappresentare tramite un diagramma a dispersione bidimensionale:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np

>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> X = df.iloc[0:100, [0, 2]].values
>>> plt.scatter(X[:50, 0], X[:50, 1],
... color='red', marker='o', label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
... color='blue', marker='x', label='versicolor')
>>> plt.xlabel('sepal length')
```

```
>>> plt.ylabel('petal length')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Eseguendo l'esempio di codice precedente, dovremmo ottenere il diagramma a dispersione rappresentato nella Figura 2.6.

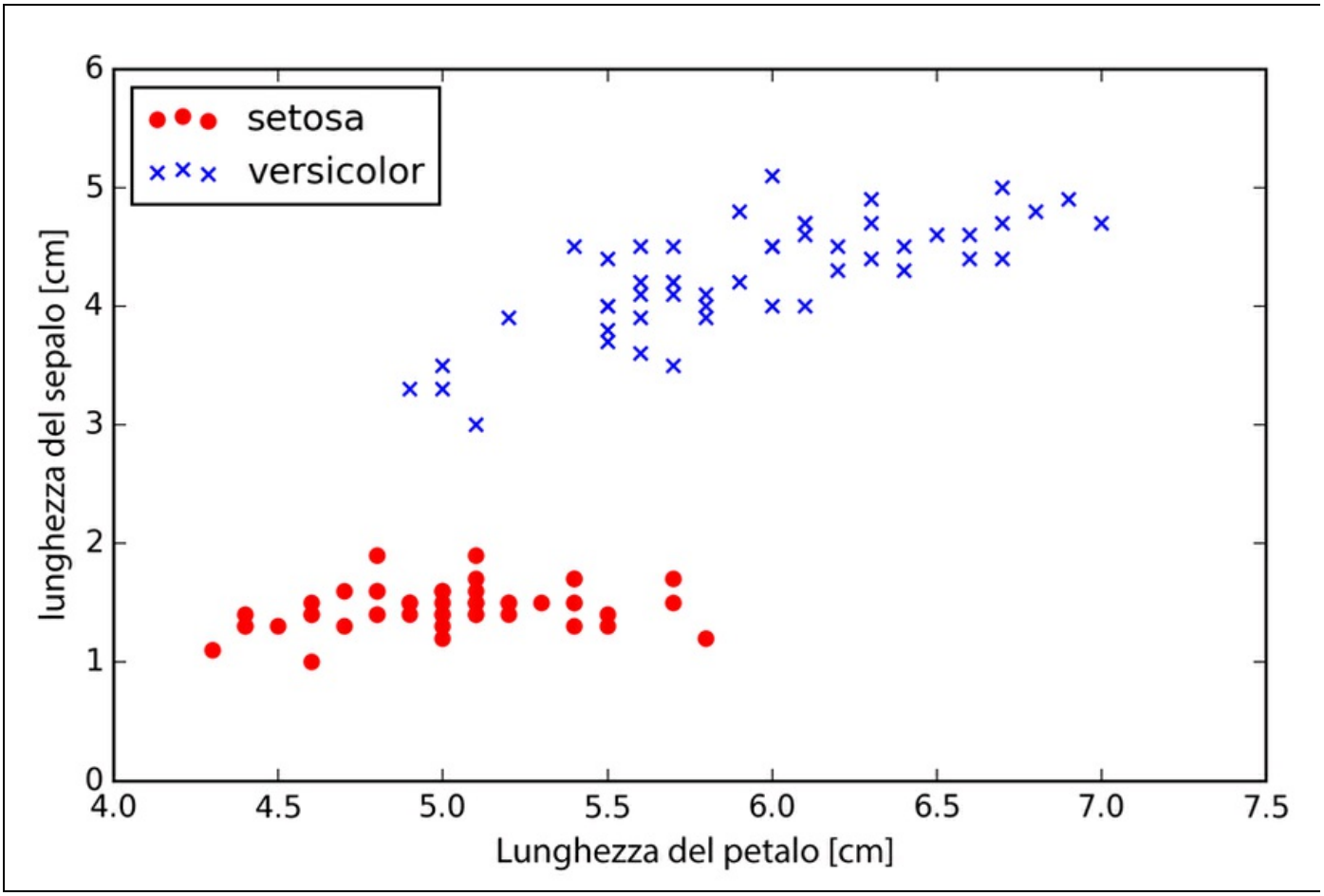


Figura 2.6

Ora è giunto il momento di addestrare il nostro algoritmo perceptron sul sottoinsieme dei dati Iris che abbiamo appena estratto. Inoltre, tracceremo l'errore di errata classificazione (*misclassification error*) per ogni epoch, per controllare che l'algoritmo converga e trovi un confine decisionale in grado di separare le due classi di fiori iris:

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_,
... marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Number of misclassifications')
>>> plt.show()
```

Dopo aver eseguito il codice precedente, otterremo il grafico degli errori di errata classificazione rispetto al numero di epoch, illustrato nella Figura 2.7.

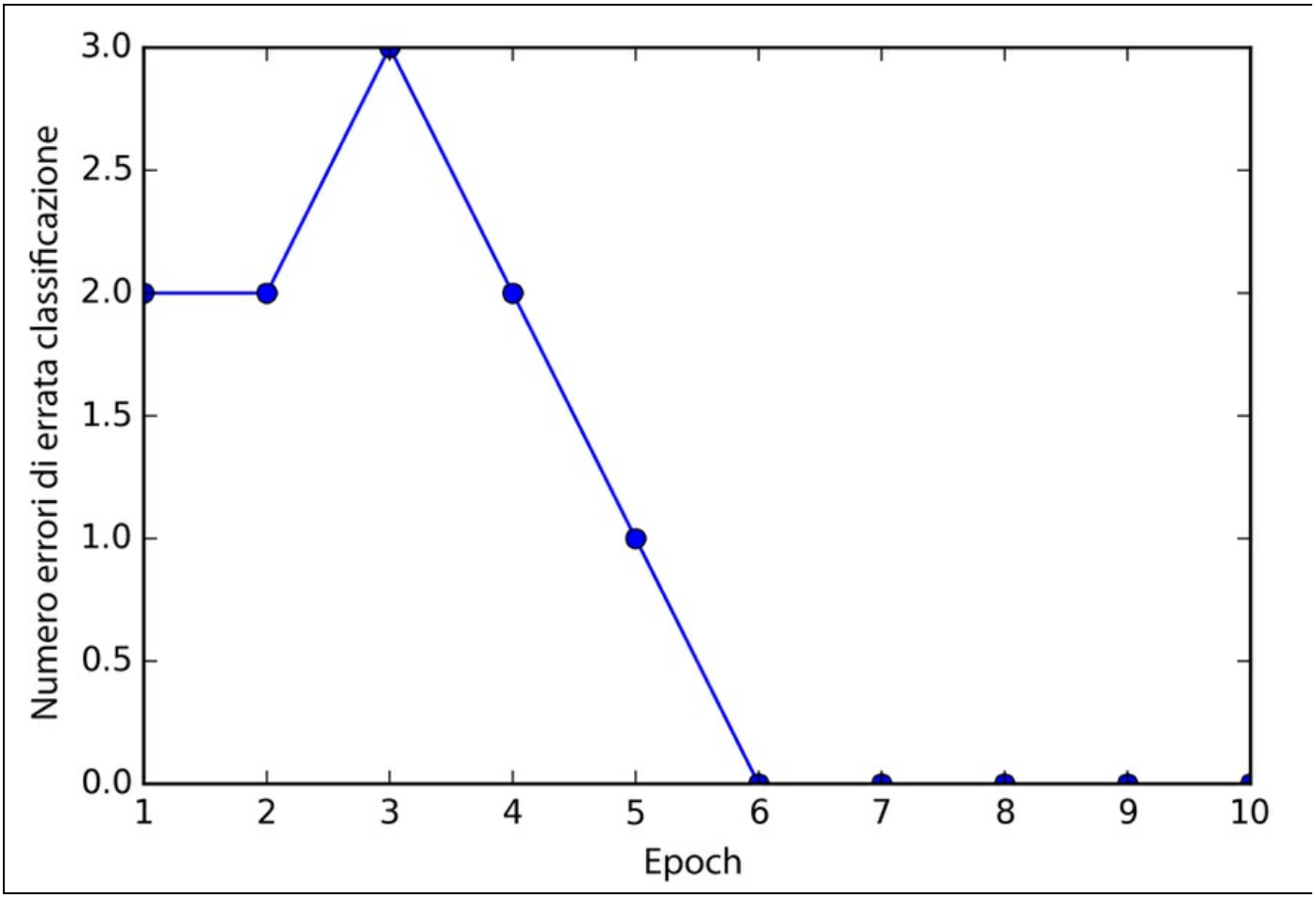


Figura 2.7

Come possiamo vedere nel grafico, il nostro perceptron converge già dopo la sesta epoch e dovrebbe essere in grado di classificare perfettamente i campioni di addestramento. Implementiamo ora una piccola funzione di utilità per rappresentare il confine decisionale per i dataset bidimensionali:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution), np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1], alpha=0.8, c=cmap(idx), marker=markers[idx], label=cl)
```

Innanzitutto definiamo un numero di colors e markers e creiamo una mappa dei colori dalla lista dei colori, tramite ListedColormap. Poi determiniamo il valore minimo e massimo per le due caratteristiche e utilizziamo i vettori delle caratteristiche per creare una copia di array a griglia xx1 e xx2 tramite la funzione NumPy meshgrid. Poiché abbiamo addestrato il nostro classificatore perceptron su due sole dimensioni delle caratteristiche, dobbiamo appiattire gli array a griglia e creare una matrice che abbia lo stesso numero di colonne del sottoinsieme di addestramento del dataset Iris, in modo da poter utilizzare il metodo predict per predire le etichette della classe z dei punti corrispondenti, sulla griglia. Dopo aver rielaborato le etichette della classe prevista z in una griglia con le stesse dimensioni di xx1 e xx2, possiamo tracciare, tramite la funzione contourf di matplotlib un grafico a contorno che mappa le diverse regioni decisionali su colori differenti per ogni classe prevista nell'array a griglia:

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Dopo aver eseguito l'esempio di codice precedente, dovremo ottenere un grafico delle regioni decisionali, come quello rappresentato nella Figura 2.8.

Come possiamo vedere dal grafico, il perceptron ha individuato un confine decisionale che è stato in grado di classificare perfettamente tutti i campioni di fiori presenti nel sottoinsieme di addestramento Iris.

NOTA

Sebbene il perceptron abbia classificato perfettamente le due classi di fiori Iris, la convergenza è uno dei problemi più gravi del perceptron. Frank Rosenblatt ha dimostrato matematicamente che la regola di apprendimento del perceptron converge se le due classi possono essere separate da un iperpiano lineare. Al contrario, se le classi non possono essere separate perfettamente da un confine decisionale così lineare, i pesi non smetteranno mai di variare, a meno che non impostiamo un numero massimo di epoch.

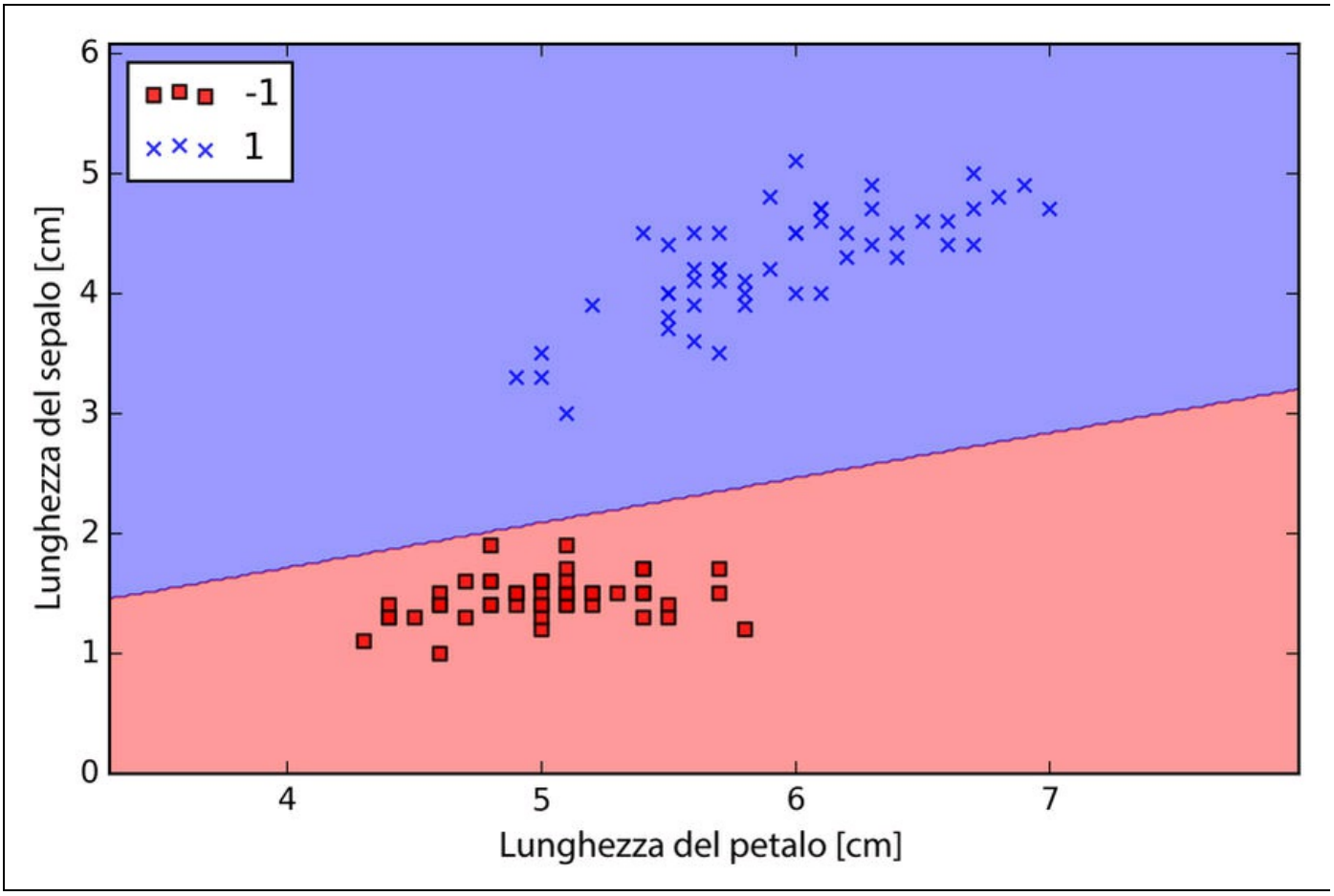

-1
1
Lunghezza del sepalo [cm]
Lunghezza del petalo [cm]
0
1
2
3
4
5
6
4
5
6
7

Figura 2.8

Neuroni adattativi lineari e convergenza dell'apprendimento

Ora esamineremo un altro tipo di rete neurale monolivello, il cui nome è *ADAptive LInear NEuron* (*Adaline*). L'algortitmo Adaline è stato pubblicato, solo pochi anni dopo l'algoritmo perceptron di Frank Rosenblatt, da Bernard Widrow e dal suo studente di dottorato Tedd Hoff e può essere considerato come un miglioramento di quest'ultimo (B. Widrow et al. Adaptive *"Adaline" neuron using chemical "memistors"*, Number Technical Report 1553-2. Stanford Electron. Labs. Stanford, CA, Ottobre 1960). L'algoritmo Adaline è particolarmente interessante, poiché illustra il concetto chiave che consiste nel definire e poi minimizzare le funzioni di costo, il che getta le basi per comprendere il funzionamento di algoritmi di classificazione più avanzati di machine learning, come quelli a regressione logistica e delle macchine vettoriale di supporto, come pure i modelli a regressione di cui parleremo nei prossimi capitoli.

La differenza principale fra la regola di Adaline (chiamata anche *regola Widrow-Hoff*) e il perceptron di Rosenblatt è il fatto che i pesi vengono aggiornati sulla base di una funzione di attivazione lineare piuttosto che su una funzione di passo unitario come avviene nel perceptron. In Adaline, questa funzione di attivazione lineare $\phi(z)$ è semplicemente la funzione identità dell'input della rete, in modo che $\phi\left(\boldsymbol{w}^T \boldsymbol{x}\right) = \boldsymbol{w}^T \boldsymbol{x}$.

Mentre la funzione di attivazione lineare viene utilizzata per l'apprendimento dei pesi, può essere impiegato un *quantizzatore*, che è simile alla funzione a passo unitario che abbiamo visto all'opera in precedenza. Il quantizzatore prevede le etichette delle classi come illustrato nella Figura 2.9.

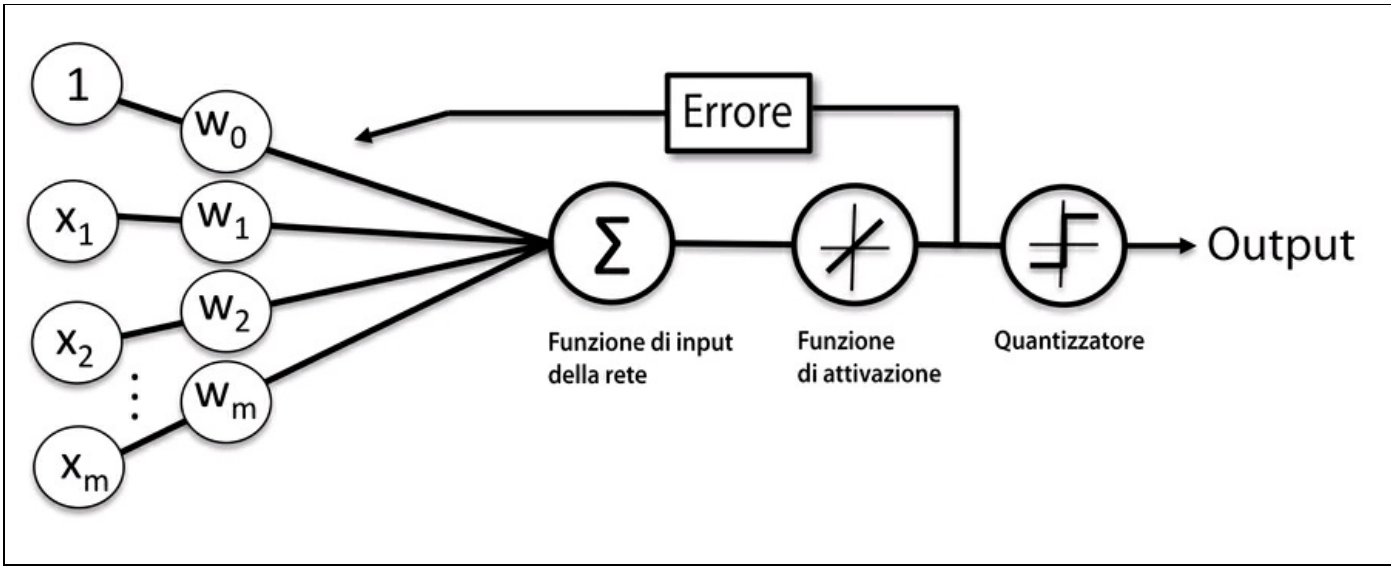


Figura 2.9

Se confrontiamo tale figura con la Figura 2.4 che rappresenta l'algoritmo perceptron, la differenza è che sappiamo utilizzare l'output continuo offerto dalla funzione ad attivazione lineare per calcolare l'errore del modello e aggiornare i pesi, invece di contare sulle etichette binarie di una classe.

Minimizzare le funzioni di costo con la discesa del gradiente

Uno degli ingredienti principali degli algoritmi di machine learning con supervisione consiste nel definire una *funzione obiettivo* che venga ottimizzata durante il processo di apprendimento. Questa funzione obiettivo è spesso una *funzione di costo*, che vogliamo minimizzare. Nel caso di Adaline, possiamo definire la funzione di costo *J* per apprendere i pesi in termini di somma dei quadrati degli errori (*Sum of Squared Errors – SSE*) fra il risultato calcolato e la vera etichetta della classe

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

Il termine $\frac{1}{2}$ è aggiunto solo per comodità; semplificherà l'operazione di derivazione del gradiente, come vedremo nei prossimi paragrafi. Il vantaggio principale di questa funzione di attivazione lineare continua è (rispetto alla funzione a passo unitario) che la funzione di costo diviene differenziabile. Un'altra ottima proprietà di questa funzione di costo è che è convessa; pertanto possiamo utilizzare

un semplice, ma potente, algoritmo di ottimizzazione chiamato discesa del gradiente (*gradient descent*) per trovare i pesi che minimizzano la nostra funzione di costo per classificare i campioni contenuti nel dataset Iris.

Come illustrato nella Figura 2.10, possiamo descrivere il principio su cui si basa la discesa del gradiente come una *discesa lungo il pendio*, fino a trovare un minimo dei costi, locale o globale. A ogni iterazione, compiamo un passo di allontanamento dal gradiente, dove l'entità del passo è determinata dal valore del tasso di apprendimento, così come dalla discesa del gradiente.

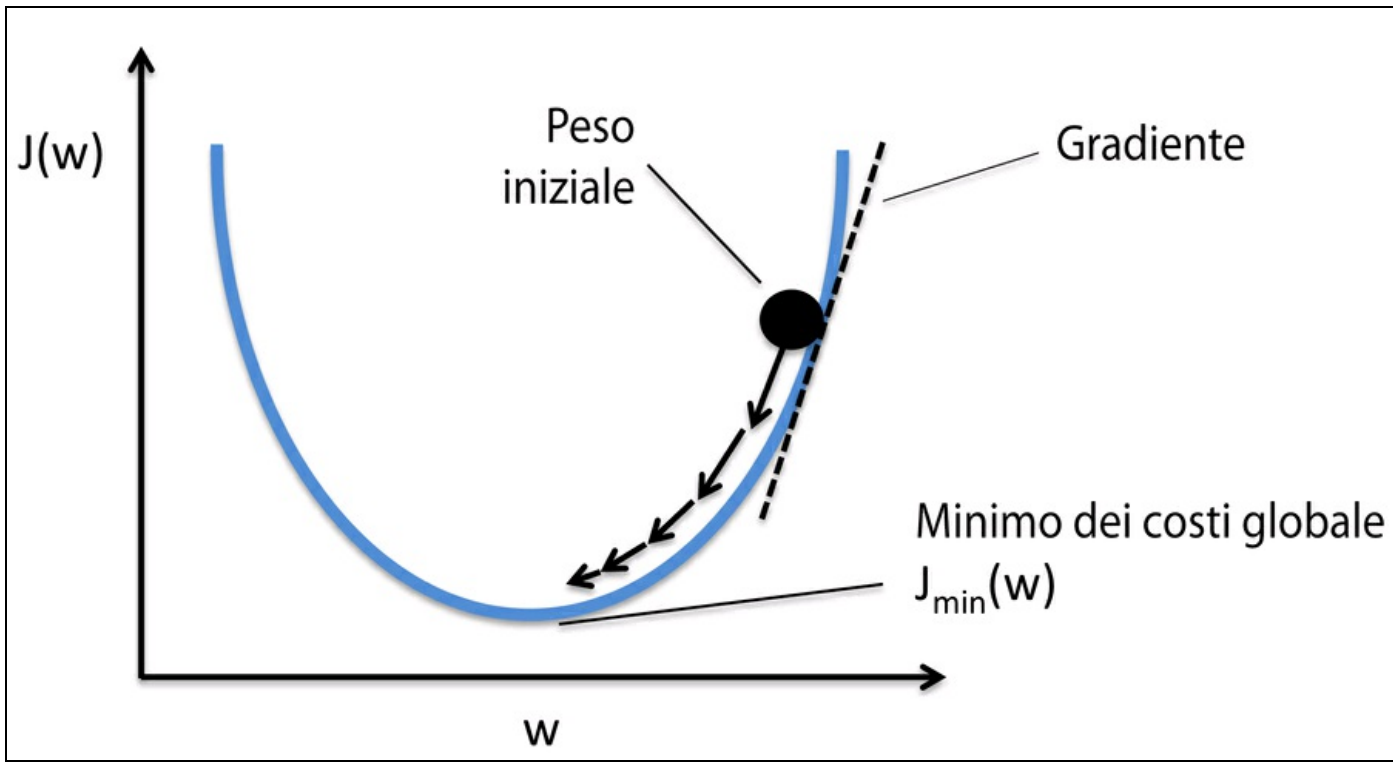


Figura 2.10

Utilizzando la tecnica della discesa del gradiente possiamo ora aggiornare i pesi, allontanandoci dal gradiente $\nabla J(w)$ della nostra funzione di costo $J(w)$:

$$w := w + \Delta w$$

Qui, il cambiamento del peso Δw è definito come il gradiente negativo moltiplicato per il tasso di apprendimento η:

$$\Delta w = -\eta \nabla J(w)$$

Per calcolare il gradiente della funzione di costo, dobbiamo calcolare la derivata parziale della funzione di costo rispetto a ogni peso w_j

$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

in modo da poter scrivere l'aggiornamento del peso w_j come

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

Poiché aggiorniamo simultaneamente tutti i pesi, la nostra regola di apprendimento Adaline diviene $\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$.

Approfondimento

Per coloro che hanno familiarità col calcolo algebrico, la derivata parziale della funzione di costo SSE rispetto al peso j-esimo può essere ottenuta nel seguente modo:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

$$= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

$$= \frac{1}{2} \sum_i 2\left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)} \right)\right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \left(-x_j^{(i)} \right)$$

$$= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

Sebbene la regola di apprendimento di Adaline sembri identica a quella del perceptron, il $\phi\left(z^{(i)}\right)$ con $z^{(i)} = \boldsymbol{w}^T \boldsymbol{x}^{(i)}$ è un numero reale e non un'etichetta intera

di classe. Inoltre, l'aggiornamento dei pesi viene calcolato sulla base di tutti i campioni del set di addestramento (invece di aggiornare i pesi in modo incrementale dopo ogni campione) e questo è il motivo per cui questo approccio viene chiamato anche a discesa del gradiente "batch", a lotti.

Implementazione di un neurone lineare adattativo in Python

Poiché la regola del perceptron e quella di Adaline sono molto simili, prenderemo l'implementazione del perceptron che abbiamo definito in precedenza e modificheremo il metodo fit in modo che i pesi vengano aggiornati minimizzando la funzione di costo tramite la discesa del gradiente:

```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
      Weights after fitting.
    errors_ : list
      Number of misclassifications in every epoch.

    """

    def __init__(self, eta=0.01, n_iter=50):
      self.eta = eta
      self.n_iter = n_iter

    def fit(self, X, y):
      """ Fit training data.

      Parameters
      ----------
      X : {array-like}, shape = [n_samples, n_features]
        Training vectors,
        where n_samples is the number of samples and
        n_features is the number of features.
      y : array-like, shape = [n_samples]
        Target values.
      Returns
      -------
      self : object

      """
      self.w_ = np.zeros(1 + X.shape[1])
      self.cost_ = []
      for i in range(self.n_iter):
        output = self.net_input(X)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
       cost = (errors**2).sum() / 2.0
        self.cost_.append(cost)
      return self

    def net_input(self, X):
      """Calculate net input"""
```

```
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

Invece di aggiornare i pesi dopo aver valutato ogni singolo campione fornito per l'addestramento, come avviene nel perceptron, calcoliamo il gradiente sulla base dell'intero dataset di addestramento tramite `self.eta * errors.sum()` per il peso-zero e tramite `self.eta * X.T.dot(errors)`per i pesi da 1 a *m*, dove `X.T.dot(errors)` è una moltiplicazione matrice-vettore fra la matrice delle caratteristiche e il vettore degli errori. Come nella precedente implementazione del perceptron, raccogliamo i valori dei costi in una lista `self.cost_` per verificare se l'algoritmo converge dopo l'addestramento.

Approfondimento

Eseguire una moltiplicazione matrice-vettore è come calcolare un prodotto vettoriale dove ogni riga della matrice viene trattata come un vettore riga. Questo approccio vettoriale rappresenta una notazione più compatta e produce un calcolo più efficiente utilizzando NumPy. Per esempio:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1\times 7 + 2\times 8 + 3\times 9 \\ 4\times 7 + 5\times 8 + 6\times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

All'atto pratico, è necessaria un po' di sperimentazione per trovare un buon tasso di apprendimento η, tale da garantire una convergenza ottimale. Pertanto, per partire scegliamo due diversi tassi di apprendimento $\eta = 0.1$ e $\eta = 0.0001$ e tracciamo le funzioni dei costi rispetto al numero di epoch, per vedere con quale efficacia l'implementazione Adaline impara dai dati di addestramento.

NOTA

Il tasso di apprendimento η, come pure il numero di epochs `n_iter`, sono detti *iperparametri* degli algoritmi di apprendimento perceptron e Adaline. Nel Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione*, esamineremo tecniche differenti per trovare automaticamente i valori dei vari iperparametri che forniscono prestazioni ottimali per il modello di classificazione.

Tracciamo il grafico dei costi rispetto al numero di epoch per i due diversi tassi di apprendimento:

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
>>> ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.cost_) + 1),
... np.log10(ada1.cost_), marker='o')
>>> ax[0].set_xlabel('Epochs')
>>> ax[0].set_ylabel('log(Sum-squared-error)')
>>> ax[0].set_title('Adaline - Learning rate 0.01')
```

```
>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.cost_) + 1),
... ada2.cost_, marker='o')
>>> ax[1].set_xlabel('Epochs')
>>> ax[1].set_ylabel('Sum-squared-error')
>>> ax[1].set_title('Adaline - Learning rate 0.0001')
>>> plt.show()
```

Come possiamo vedere nei grafici rappresentati nella Figura 2.11, incontriamo due diversi tipi di problemi. Il grafico a sinistra mostra ciò che accade se scegliamo un tasso di apprendimento troppo ampio: invece di minimizzare la funzione di costo, l'errore diviene sempre più ampio a ogni epoch, poiché abbiamo esagerato (*overshoot*) con il minimo globale.

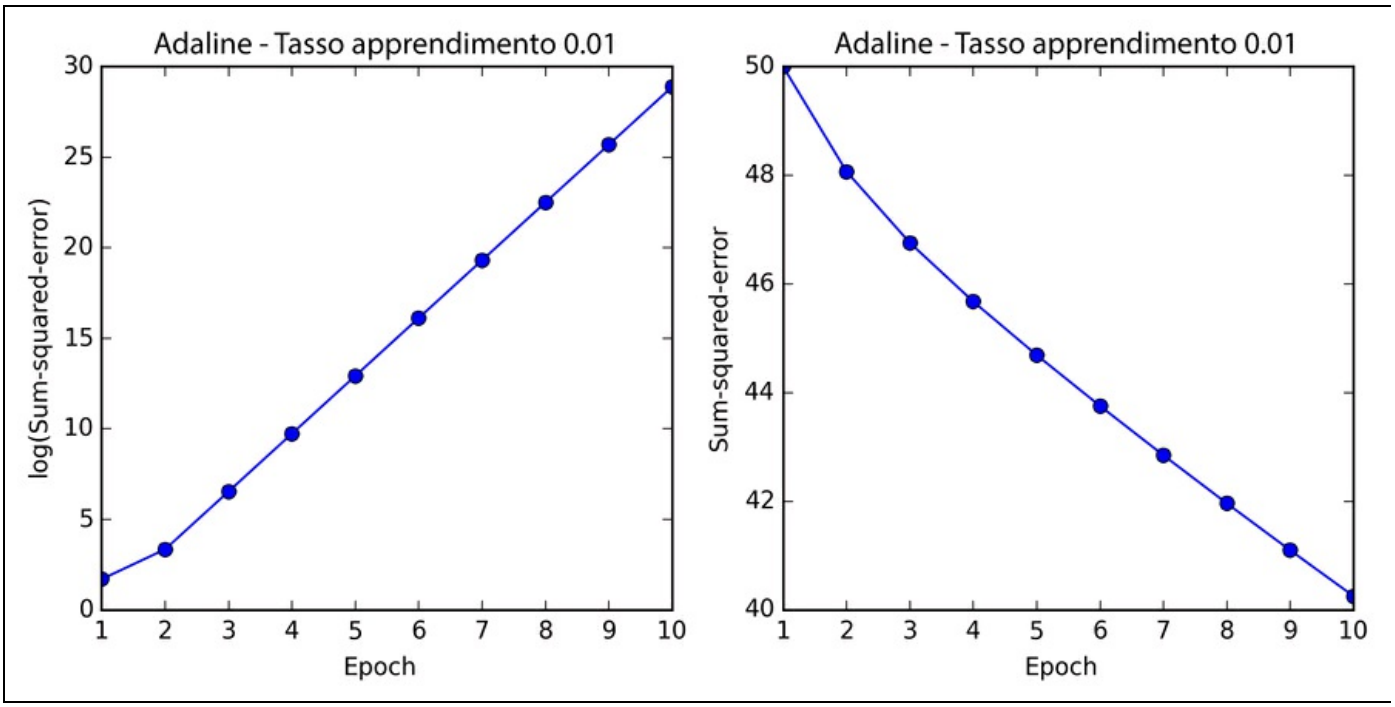


Figura 2.11

Sebbene nel grafico a destra possiamo vedere che i costi decrescono, il tasso di apprendimento scelto, $\eta = 0.0001$, è troppo piccolo e costringe l'algoritmo a convergere solo dopo un numero eccessivo di epoch. La Figura 2.12 illustra come possiamo cambiare il valore del parametro del peso per minimizzare la funzione di costo J (lato sinistro della figura). Il lato destro della figura illustra ciò che accade se scegliamo un tasso di apprendimento troppo ampio: manchiamo sempre il minimo globale.

Molti algoritmi di machine learning che incontreremo nel corso di questo libro richiedono un adattamento di scala per garantire prestazioni ottimali, argomento che tratteremo più in dettaglio nel Capitolo 3, *I classificatori di machine learning di scikit-learn*. La discesa del gradiente è uno dei tanti algoritmi che traggono vantaggi dalla riduzione in scala.

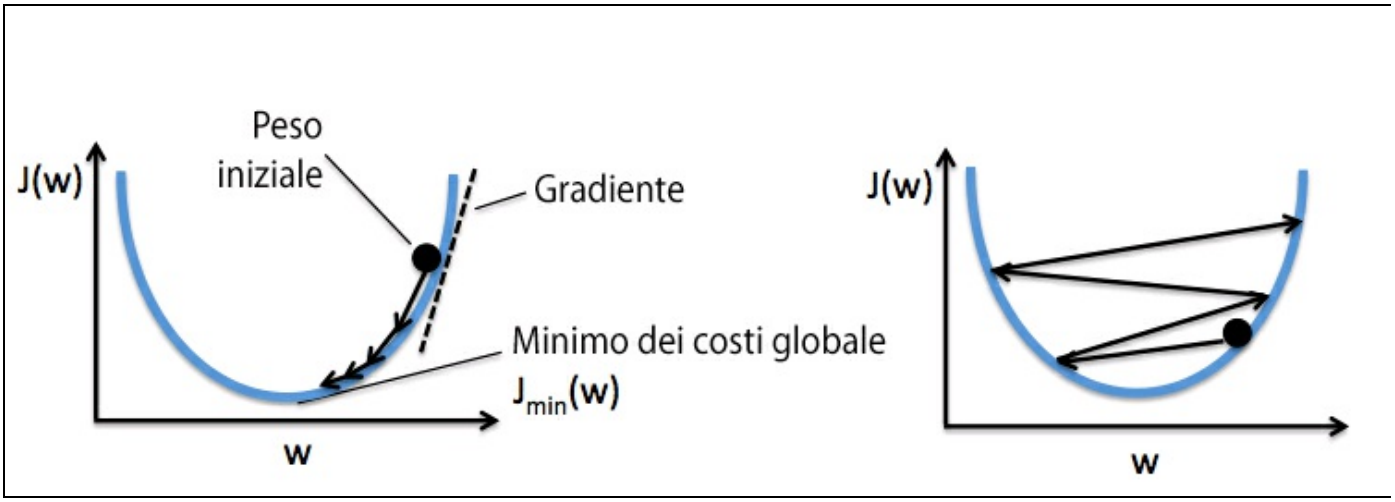


Figura 2.12

Qui utilizzeremo un metodo di riduzione in scala delle caratteristiche chiamato *standardizzazione*, che fornisce ai dati la proprietà di una distribuzione normale standard. La media di ogni caratteristica viene centrata sul valore zero e la colonna della caratteristica ha una deviazione standard pari a 1. Per esempio, per standardizzare la caratteristica j-esima, dobbiamo semplicemente sottrarre la media del campione μ_j da ogni campione di addestramento e dividere il tutto per la sua deviazione standard σ_j:

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

Qui x_j è un vettore costituito dai valori della j-esima caratteristica di tutti i campioni di addestramento n.

La standardizzazione può essere ottenuta con facilità utilizzando i metodi NumPy `mean` e `std`:

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

Dopo la standardizzazione, addestreremo nuovamente Adaline e vedremo come ora converge utilizzando un tasso di apprendimento $\eta = 0.01$:

```
>>> ada = AdalineGD(n_iter=15, eta=0.01)
>>> ada.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline - Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')
>>> plt.ylabel('petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Sum-squared-error')
>>> plt.show()
```

Dopo aver eseguito il codice precedente, dovremmo vedere una figura delle regioni decisionali e un grafico della riduzione dei costi, del tipo indicato nella Figura 2.13.

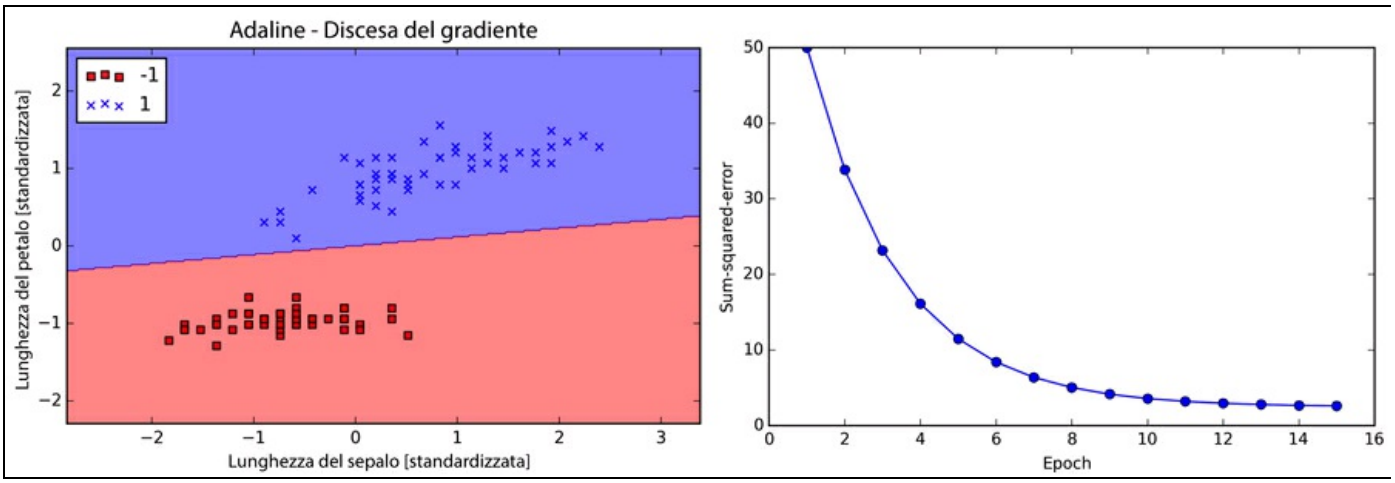


Figura 2.13

Come possiamo vedere eseguendo un confronto con i grafici precedenti, ora Adaline converge dopo l'addestramento sulle caratteristiche standardizzate utilizzando un tasso di apprendimento $\eta = 0.01$. Tuttavia, notate che l'SSE rimane diverso da zero, anche se tutti i campioni sono stati classificati correttamente.

Machine learning su larga scala e discesa stocastica del gradiente

Nel paragrafo precedente, abbiamo imparato a minimizzare una funzione di calcolo dei costi compiendo un passo nella direzione opposta rispetto a un gradiente che viene calcolato sulla base dell'intero insieme dei dati di addestramento. Questo è il motivo per cui questo approccio viene talvolta chiamato discesa del gradiente *batch*. Ora immaginate di avere un dataset molto esteso, contenente milioni di punti, il che è piuttosto comune nelle applicazioni di machine learning. L'esecuzione dell'algoritmo a discesa del gradiente batch può essere piuttosto costosa dal punto di vista computazionale, per il fatto che tale situazione costringe a rivalutare l'intero dataset di addestramento ogni volta che compiamo un passo verso il minimo globale.

Un'alternativa comune all'algoritmo a discesa del gradiente batch è quello di *discesa del gradiente stocastica*, chiamata anche *discesa del gradiente iterativa* o *online*. Invece di aggiornare i pesi sulla base della somma degli errori accumulati rispetto ai campioni $\boldsymbol{x}^{(i)}$,

$$\Delta \boldsymbol{w} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) \boldsymbol{x}^{(i)},$$

aggiorniamo i pesi in modo incrementale per ogni campione di addestramento:

$$\eta \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) \boldsymbol{x}^{(i)}$$

Sebbene la discesa del gradiente stocastica possa essere considerata un'approssimazione della discesa del gradiente, in genere raggiunge la convergenza molto più velocemente, poiché i pesi vengono aggiornati più frequentemente. Poiché ciascun gradiente viene calcolato sulla base di un singolo esempio di addestramento, la superficie di errore è più rumorosa rispetto alla discesa del gradiente, il che presenta anche avere il vantaggio che la discesa del gradiente stocastica è in grado di sfuggire con maggiore facilità agli avvallamenti rappresentati dai minimi locali. Per ottenere risultati accurati tramite la discesa del gradiente stocastica, è importante presentarle i dati in modo il più possibile casuale, motivo per cui è opportuno cambiare l'ordine del dataset a ogni epoch, per impedire l'innescarsi di cicli.

Approfondimento

Nelle implementazioni della discesa del gradiente stocastica, il tasso fisso di apprendimento η viene frequentemente sostituito da un tasso di apprendimento adattativo che si riduce nel corso del tempo, per esempio

$$\frac{c_1}{[numero\ di\ iterazioni] + c_2}$$

dove c_1 e c_2 sono costanti. Si noti che la discesa del gradiente stocastica non raggiunge il minimo globale, ma una zona che gli si avvicina molto. Utilizzando un tasso di apprendimento adattativo, possiamo ottenere un ulteriore affinamento, per individuare un minimo globale migliore.

Un altro vantaggio della discesa del gradiente stocastica è che possiamo utilizzarla per l'*apprendimento online*. Nell'apprendimento online, il nostro modello viene addestrato "al volo", mano a mano che arrivano nuovi dati di apprendimento. Questo è particolarmente utile se si stanno accumulando grandi quantità di dati, per esempio i dati sulla clientela di una tipica applicazione web. Utilizzando l'apprendimento online, il sistema può adattarsi immediatamente ai cambiamenti e i dati di addestramento possono essere eliminati dopo aver

aggiornato il modello, soluzione utile specialmente quando vi sono problemi in termini di disponibilità di spazio.

NOTA

Un compromesso fra discesa del gradiente batch e discesa del gradiente stocastica è il cosiddetto *apprendimento mini-batch*. L'apprendimento mini-batch può essere compreso come l'applicazione di una discesa del gradiente batch al piccolo sottoinsieme dei dati di addestramento, per esempio 50 campioni alla volta. Il vantaggio rispetto alla discesa del gradiente batch è il fatto che la convergenza viene raggiunta più velocemente tramite i mini-batch, a causa dell'aggiornamento più frequente dei pesi. Inoltre, l'apprendimento mini-batch consente di sostituire il ciclo *for* sui campioni di addestramento dell'algoritmo a discesa del gradiente stocastica (*Stochastic Gradient Descent – SGD)* con operazioni vettoriali, in grado di migliorare ulteriormente l'efficienza computazionale dell'algoritmo di apprendimento.

Poiché abbiamo già implementato la regola di apprendimento Adaline utilizzando la discesa del gradiente, dobbiamo solo applicare alcuni interventi per modificare l'algoritmo di apprendimento in modo che aggiorni i pesi tramite la discesa del gradiente stocastica. All'interno del metodo `fit`, aggiorneremo i pesi dopo ciascun campione di addestramento. Inoltre, implementeremo un ulteriore metodo `partial_fit`, che non reinizializza i pesi, per l'apprendimento online. Con lo scopo di verificare se il nostro algoritmo converge dopo l'addestramento, calcoleremo il costo in termini di costo medio dei campioni di addestramento in ciascuna epoch. Inoltre, aggiungeremo un'opzione per mescolare (`shuffle`) i dati di addestramento prima di ogni epoch, per evitare l'insorgere di cicli durante l'ottimizzazione della funzione del costo. Tramite il parametro `random_state` consentiamo di specificare un seme casuale per la coerenza:

```
from numpy.random import seed
class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
      Weights after fitting.
    errors_ : list
      Number of misclassifications in every epoch.
    shuffle : bool (default: True)
      Shuffles training data every epoch
      if True to prevent cycles.
    random_state : int (default: None)
      Set random state for shuffling
      and initializing the weights.

    """

    def __init__(self, eta=0.01, n_iter=10,
                 shuffle=True, random_state=None):
```

```
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        if random_state:
            seed(random_state)

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples
            is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.
        Returns
        -------
        self : object

         """
        self._initialize_weights(X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost)/len(y)
            self.cost_.append(avg_cost)
        return self

    def partial_fit(self, X, y):
        """Fit training data without reinitializing the weights"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Shuffle training data"""
        r = np.random.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Initialize weights to zeros"""
        self.w_ = np.zeros(1 + m)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Apply Adaline learning rule to update the weights"""
        output = self.net_input(xi)
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
        cost = 0.5 * error**2
        return cost

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

Il metodo _shuffle che utilizziamo ora nel classificatore AdalineSGD funziona nel modo seguente: tramite la funzione permutation in numpy.random, generiamo una sequenza casuale di numeri univoci nell'intervallo compreso fra 0 e 100. Questi numeri possono poi essere utilizzati come indici per mescolare la matrice delle caratteristiche e il vettore delle etichette delle classi.

Possiamo poi utilizzare il metodo fit per addestrare il classificatore AdalineSGD e utilizzare il nostro plot_decision_regions per tracciare i risultati dell'addestramento:

```
>>> ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline - Stochastic Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')
>>> plt.ylabel('petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Average Cost')
>>> plt.show()
```

I due grafici che otteniamo dall'esecuzione del codice precedente sono rappresentati nella Figura 2.14.

Come possiamo vedere, il costo medio si abbatte con buona rapidità e la decisione finale sul confine dopo quindici epoch ha lo stesso aspetto della discesa del gradiente batch con Adaline. Se vogliamo aggiornare il nostro modello (per esempio in una situazione di apprendimento online, con dati che giungono in streaming), ci basta richiamare il metodo partial_fit sui singoli campioni, come in ada.partial_fit(X_std[0, :], y[0]).

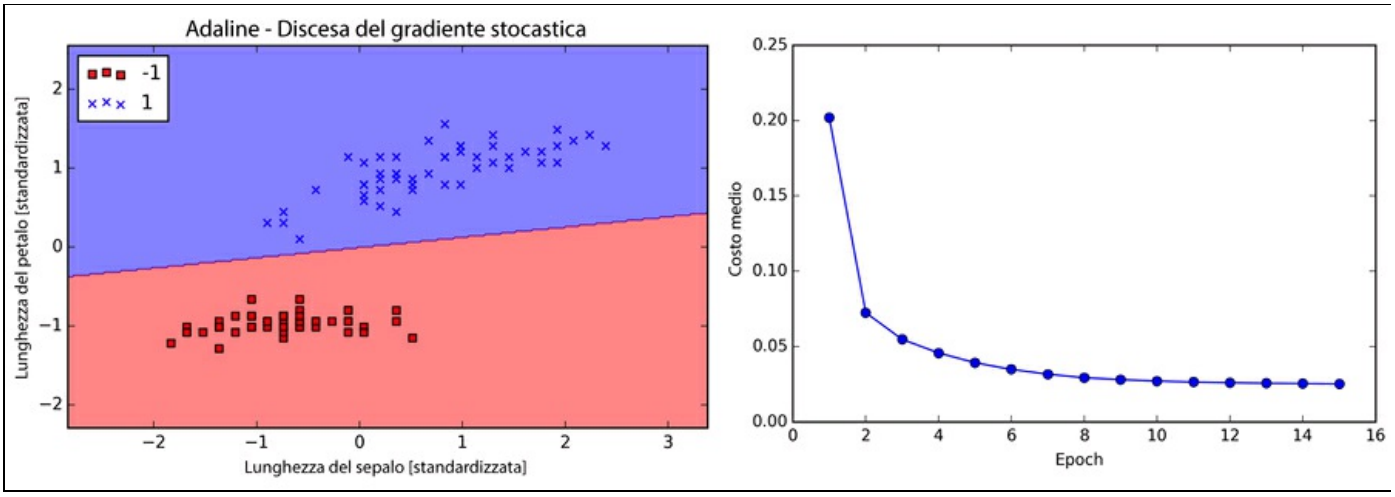


Figura 2.14

Riepilogo

In questo capitolo abbiamo trattato i concetti di base dei classificatori lineari per l'apprendimento con supervisione. Dopo abbiamo implementato un perceptron, abbiamo visto come possiamo addestrare i neuroni lineari adattativi in modo efficiente tramite un'implementazione vettoriale della discesa del gradiente e abbiamo parlato di apprendimento online tramite la discesa del gradiente stocastica.

Ora che abbiamo visto come implementare semplici classificatori in Python, siamo pronti per passare al capitolo successivo, dove utilizzeremo la libreria di machine learning Python, scikit-learn, per ottenere l'accesso a classificatori più avanzati e potenti, già pronti all'uso, comunemente utilizzati sia in ambito accademico, sia in campo professionale.

I classificatori di machine learning di scikit-learn

In questo capitolo effettueremo un tour sui più noti e potenti algoritmi di machine learning comunemente impiegati in ambito accademico e professionale. Mentre studieremo le differenze esistenti fra i vari algoritmi di apprendimento con supervisione dedicati alla classificazione, svilupperemo anche una valutazione intuitiva delle loro capacità e dei loro punti deboli. Inoltre, compieremo i primi passi nell'impiego della libreria scikit-learn, che offre un'interfaccia di facile uso per l'impiego di questi algoritmi, in modo efficiente e produttivo.

Gli argomenti che impareremo nel corso di questo capitolo sono i seguenti.

- Introduzione ai concetti dei più noti algoritmi di classificazione.
- Uso della libreria di machine learning scikit-learn.
- Quali domande è opportuno porsi per scegliere l'algoritmo più appropriato di machine learning.

Scelta di un algoritmo di classificazione

La scelta di un algoritmo di classificazione appropriato per risolvere un determinato problema richiede pratica: ogni algoritmo ha le sue specificità e si basa su determinati presupposti. Ribadiamo il concetto che "non esiste cibo gratis": nessuno dei classificatori opera al meglio in tutte le situazioni possibili. In pratica, è sempre consigliabile confrontare le prestazioni di almeno una manciata di algoritmi di apprendimento differenti, per selezionare quello migliore, in grado di risolvere uno specifico problema; i problemi possono differire in termini di numero di caratteristiche o campioni, quantità di rumore nel dataset e per il fatto che le classi siano separabili in modo lineare oppure no.

Alla fine, le prestazioni di un classificatore, la sua potenza computazionale e la sua capacità predittiva dipendono in larga misura dai dati che sono disponibili per l'apprendimento. I cinque passi principali che sono coinvolti nell'addestramento di un algoritmo di machine learning e possono essere riepilogati nel seguente modo.

1. Scelta delle caratteristiche.
2. Scelta di una metrica prestazionale.
3. Scelta di un algoritmo classificatore e di ottimizzazione.
4. Valutazione delle prestazioni del modello.
5. Ottimizzazione dell'algoritmo.

Poiché l'approccio di questo libro consiste nel costruire le competenze di machine learning passo dopo passo, in questo capitolo ci concentreremo principalmente sui concetti fondamentali dei vari algoritmi ed esamineremo argomenti quali la scelta delle caratteristiche e la pre-elaborazione, le metriche prestazionali e l'ottimizzazione degli iperparametri; tali argomenti verranno poi affinati nel corso dei prossimi capitoli.

Primi passi con scikit-learn

Nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione,* abbiamo trattato due algoritmi di classificazione correlati fra loro: la regola del perceptron e Adaline, che abbiamo implementato autonomamente in Python. Ora esamineremo l'API scikit-learn, che combina una comoda interfaccia utente e un'implementazione molto ottimizzata di numerosi algoritmi di classificazione. Tuttavia, la libreria scikit-learn non offre solo una grande varietà di algoritmi di apprendimento, ma anche molte comode funzioni di pre-elaborazione dei dati e di ottimizzazione e valutazione dei modelli. Ne tratteremo più in dettaglio, parlando anche dei concetti su cui si basano, nel Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione*, e nel Capitolo 5, *Compressione dei dati tramite la riduzione della dimensionalità*.

Addestramento di un perceptron tramite scikit-learn

Per iniziare a impiegare la libreria scikit-learn, addestreremo un modello di perceptron simile a quello che abbiamo implementato nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*. Per semplicità, nei prossimi paragrafi utilizzeremo il dataset Iris, che ormai dovrebbe essere familiare. Per comodità, il dataset Iris è già disponibile tramite scikit-learn, in quanto si tratta di un dataset semplice e molto noto, che viene frequentemente utilizzato per le attività di collaudo e sperimentazione con gli algoritmi. Inoltre utilizzeremo due caratteristiche del dataset di fiori Iris per compiti di visualizzazione.

Assegneremo la *lunghezza del petalo* e la *larghezza del petalo* dei 150 campioni di fiori alla matrice delle caratteristiche `X` e le corrispondenti etichette di classificazione delle specie di fiori al vettore `y`:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
```

Se eseguissimo `np.unique(y)` per restituire le varie etichette di classificazione conservate in `iris.target`, vedremmo che i nomi delle classi di fiori, *Iris-Setosa*, *Iris-Versicolor* e *Iris-Virginica*, sono già conservati sotto forma di numeri interi (`0`, `1`, `2`), una pratica consigliata per garantire prestazioni ottimali di molte librerie di machine learning.

Per valutare la qualità di un modello operante su dati sconosciuti, suddivideremo ulteriormente il dataset in due dataset: di addestramento e di test. Successivamente, nel Capitolo 5, *Compressione dei dati tramite la riduzione della dimensionalità*, tratteremo più in dettaglio le migliori pratiche per la valutazione dei modelli:

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...         X, y, test_size=0.3, random_state=0)
```

Utilizzando la funzione `train_test_split` del modulo `cross_validation` di scikit-learn, suddividiamo casualmente gli array `X` e `y` in un 30% di dati di test (45 campioni) e un 70% di dati di addestramento (105 campioni).

Molti algoritmi di machine learning e di ottimizzazione richiedono anche una riduzione di scala per garantire prestazioni ottimali, come abbiamo visto nell'esempio dell'algoritmo a discesa del gradiente del Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*. Qui, standardizzeremo le caratteristiche utilizzando la classe `StandardScaler` del modulo `preprocessing` di scikit-learn:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Utilizzando il codice precedente, abbiamo caricato la classe `StandardScaler` dal modulo di pre-elaborazione e abbiamo inizializzato un nuovo oggetto `StandardScaler` che abbiamo assegnato alla variabile `sc`. Utilizzando il metodo `fit`, `StandardScaler` ha stimato i parametri μ (media del campione) e σ (deviazione standard) per ciascuna dimensione delle caratteristiche presenti nei dati di addestramento. Richiamando il metodo `transform`, abbiamo poi standardizzato i dati di addestramento utilizzando questi parametri stimati, μ e σ. Notate che abbiamo utilizzato gli stessi parametri di riduzione in scala per standardizzare il dataset di test, in modo che i valori del dataset di addestramento e del dataset di test fossero confrontabili.

Avendo standardizzato i dati di addestramento, possiamo ora addestrare il modello del perceptron. La maggior parte degli algoritmi di scikit-learn supporta già la classificazione multiclasse, tramite il metodo *One-vs.-Rest* (*OvR*), il che ci consente di fornire al perceptron, insieme, tutte e tre le classi di fiori.

Il codice è il seguente:

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)
>>> ppn.fit(X_train_std, y_train)
```

L'interfaccia di scikit-learn ci ricorda la nostra implementazione del perceptron svolta nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*: dopo

aver caricato la classe Perceptron dal modulo linear_model, abbiamo inizializzato un nuovo oggetto Perceptron e abbiamo addestrato il modello tramite il metodo fit. Qui, il parametro del modello eta0 è equivalente al tasso di apprendimento eta che abbiamo utilizzato nella nostra implementazione del perceptron e il parametro n_iter definisce il numero di epoch (passi sul set di addestramento). Come possiamo ricordare dal Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, la ricerca di un tasso di apprendimento appropriato richiede un po' di sperimentazione. Se il tasso di apprendimento è eccessivo, l'algoritmo mancherà il minimo costo globale. Se invece il tasso di apprendimento è troppo ridotto, l'algoritmo richiederà troppe epoch prima di convergere, il che può rallentare l'apprendimento, specialmente per dataset di grandi dimensioni. Inoltre, abbiamo utilizzato il parametro random_state per la riproducibilità del mescolamento iniziale del dataset di addestramento dopo ogni epoch.

Dopo aver addestrato un modello in scikit-learn, possiamo eseguire previsioni tramite il metodo predict, così come avveniva nella nostra implementazione del perceptron nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*. Il codice è il seguente:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified samples: %d' % (y_test != y_pred).sum())
Misclassified samples: 4
```

Nell'esecuzione del codice precedente, vediamo che il perceptron fallisce nel classificare 4 campioni di fiori su 45. Pertanto, l'errore di errata classificazione del dataset di test è pari a 0.089, ovvero pari 8.9% $(4 / 45 \approx 0.089)$.

NOTA

Invece dell'errore di errata classificazione, molti esperti di machine learning parlano di accuratezza di classificazione di un modello, che si calcola semplicemente nel seguente modo:

1 - *errore di errata classificazione* = 0,911% o 91,1%

scikit-learn implementa anche un'ampia varietà di metriche prestazionali, disponibili tramite il modulo metrics. Per esempio, possiamo calcolare l'accuratezza di classificazione del perceptron sul set di test nel seguente modo:

```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
0.91
```

Qui, y_test sono le vere etichette delle classi e y_pred sono le etichette delle classi che abbiamo appena previsto.

NOTA

Notate che valutiamo le prestazioni dei nostri modelli sulla base del dataset di test di questo capitolo. Nel Capitolo 5, *Compressione dei dati tramite la riduzione della dimensionalità*,

impareremo utili tecniche, fra cui tecniche di analisi grafica come le curve di apprendimento, per rilevare e impedire il problema dell'overfitting. Si ha un *overfitting* quando il modello cattura bene gli schemi durante l'addestramento, ma non riesce a generalizzare tali risultati su dati sconosciuti.

Infine, possiamo utilizzare la nostra funzione plot_decision_regions del Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, per tracciare le regioni decisionali del modello di perceptron che abbiamo appena addestrato e rappresentare la qualità con cui separa i vari campioni di fiori. Tuttavia, introduciamo una piccola modifica per evidenziare i campioni del dataset di test tramite piccoli cerchi:

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
def plot_decision_regions(X, y, classifier,
                          test_idx=None, resolution=0.02):
    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])
    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
    # plot all samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)

 # highlight test samples
 if test_idx:
 X_test, y_test = X[test_idx, :], y[test_idx]
 plt.scatter(X_test[:, 0], X_test[:, 1], c='',
 alpha=1.0, linewidths=1, marker='o',
             s=55, label='test set')
```

Con questa lieve modifica che abbiamo apportato alla funzione plot_decision_regions (evidenziata nel codice precedente), possiamo ora specificare gli indici dei campioni che vogliamo contrassegnare sui grafici risultanti. Il codice è il seguente:

```
>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                       y=y_combined,
...                       classifier=ppn,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Come possiamo vedere nel grafico rappresentato nella Figura 3.1, le tre classi di fiori non riescono a essere separate perfettamente da confini decisionali puramente lineari.

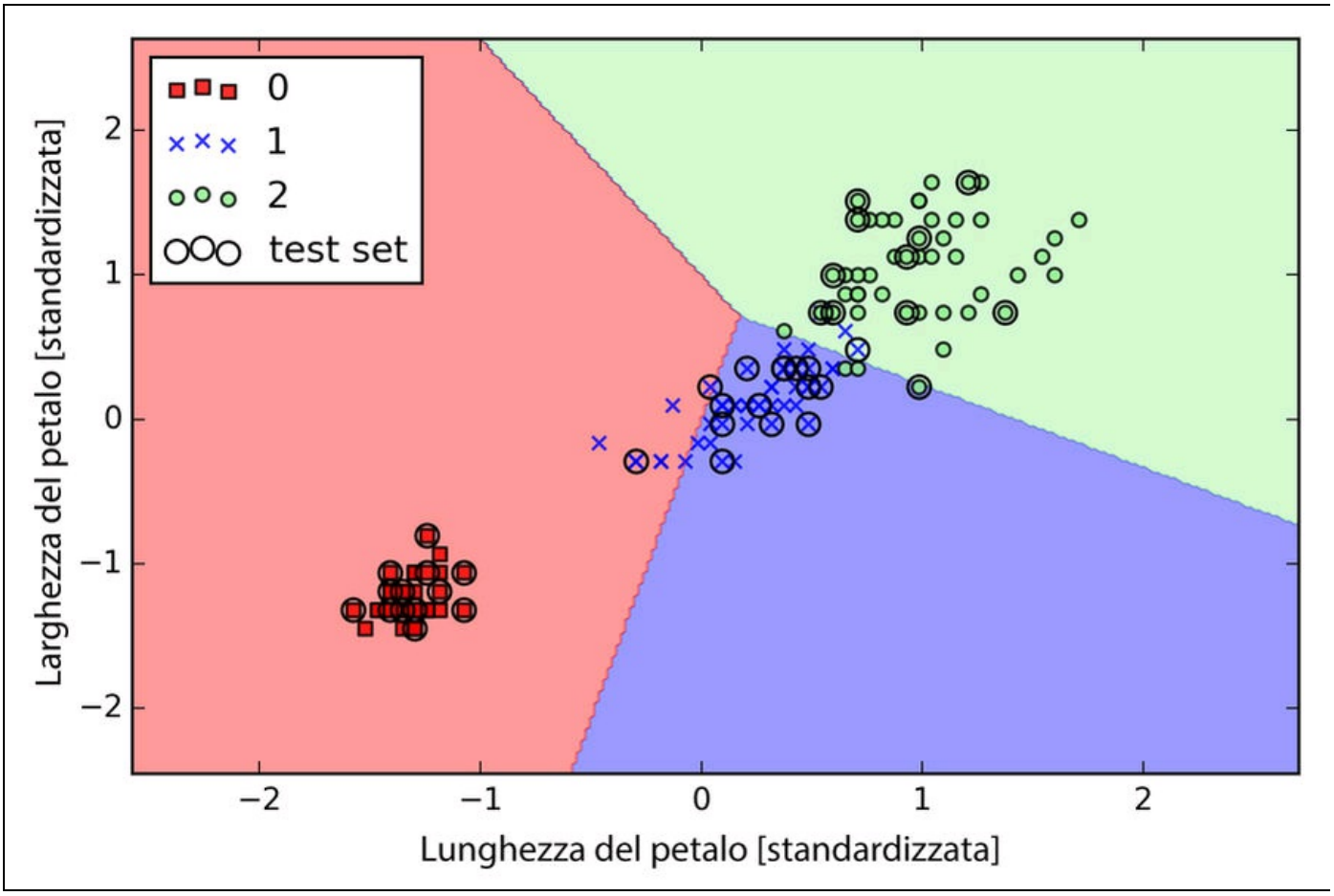


Figura 3.1

Dalla precedente discussione fatta nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, ricordiamo che l'algoritmo perceptron non converge mai su dataset che non siano separabili perfettamente in modo lineare; questo è il motivo per cui l'utilizzo dell'algoritmo perceptron, normalmente, è sconsigliato nell'attività pratica. Nei prossimi paragrafi, esamineremo classificatori lineari più potenti, che convergono a un costo minimo anche nel caso in cui le classi non siano perfettamente separabili in modo lineare.

NOTA

Il `Perceptron`, così come altre funzioni e classi di scikit-learn, ha anche altri parametri, che qui abbiamo omesso per semplicità. Per informazioni su questi parametri, utilizzate la funzione `help` di Python (per esempio `help(Perceptron)`) oppure consultate l'eccellente documentazione online disponibile su scikit-learn all'indirizzo http://scikit-learn.org/stable/.

Modellazione delle probabilità delle classi tramite la regressione logistica

Sebbene la regola del perceptron offra un'introduzione pratica e semplice agli algoritmi di machine learning per la classificazione, il suo grande svantaggio è che non converge mai quando le classi non siano perfettamente separabili in modo lineare. Il compito di classificazione del paragrafo precedente è proprio un esempio di questa situazione. Intuitivamente, possiamo comprendere il motivo per cui i pesi cambiano continuamente, in quanto in ciascuna epoch vi è sempre almeno un campione erroneamente classificato. Naturalmente, è sempre possibile cambiare il tasso di apprendimento e incrementare il numero di epoch, ma sappiamo già che il perceptron non potrà mai convergere su questo dataset. Per utilizzare meglio il nostro tempo, esamineremo un altro algoritmo, semplice ma più potente, per la classificazione di problemi lineari e binari: la *regressione logistica*. Notate che, nonostante il suo nome, l'algoritmo a regressione logistica è un modello di classificazione e non di regressione.

Concetti intuitivi e probabilità condizionali della regressione logistica

L'algoritmo a regressione logistica è un modello di classificazione che è molto facile da implementare, ma si comporta molto bene su classi che possono essere separate linearmente. È uno degli algoritmi più ampiamente utilizzati per la classificazione. In modo analogo al perceptron e ad Adaline, il modello a regressione logistica di questo capitolo è un modello lineare per la classificazione binaria, che può essere esteso alla classificazione multiclasse tramite una tecnica OvR.

Per spiegare gli elementi di base della regressione logistica come modello probabilistico, occorre innanzitutto introdurre il rapporto probabilistico, ovvero le probabilità favorevoli relative a un determinato evento. Il rapporto probabilistico può essere scritto come

$$\frac{p}{(1-p)}$$

dove p è la probabilità dell'evento positivo. Il termine *evento positivo* non significa necessariamente *buono*, ma fa riferimento all'evento che vogliamo prevedere, per esempio la probabilità che un paziente sia affetto da una determinata malattia; possiamo pensare a un evento positivo come l'etichetta della classe $y = 1$. Possiamo poi ulteriormente definire la funzione *logit*, che è semplicemente il logaritmo del rapporto probabilistico (log-odds):

$$logit(p) = log \frac{p}{(1-p)}$$

La funzione logit accetta valori di input nell'intervallo fra 0 e 1 e li trasforma in valori lungo l'intero intervallo di numeri reali, che possiamo utilizzare per esprimere una relazione lineare fra i valori delle caratteristiche e log-odds:

$$logit\left(p\left(y=1 \mid \boldsymbol{x}\right)\right) = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{i=0}^{n} w_m x_m = \boldsymbol{w}^T \boldsymbol{x}$$

Qui, $p(y=1 \mid \boldsymbol{x})$ è la probabilità condizionale che un determinato campione appartenga alla classe 1 sulla base delle sue caratteristiche *x*.

Ora, ciò che ci interessa davvero è prevedere la probabilità che un determinato campione appartenga a una determinata classe, che è la forma inversa della funzione logit. È chiamata anche funzione *logistica*, talvolta abbreviata semplicemente come funzione *sigmoid*, a causa della sua caratteristica forma a "S".

$$\phi(z) = \frac{1}{1+e^{-z}}$$

Qui, *z* è l'input della rete, ovvero la combinazione lineare dei pesi e delle caratteristiche dei campioni e può essere calcolata come $z = \boldsymbol{w}^T \boldsymbol{x} = w_0 + w_1 x_1 + \cdots + w_m x_m$.

Ora tracciamo semplicemente la funzione sigmoid per alcuni valori compresi nell'intervallo fra -7 e 7 per vedere il loro aspetto:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> phi_z = sigmoid(z)
>>> plt.plot(z, phi_z)
>>> plt.axvline(0.0, color='k')
>>> plt.axhspan(0.0, 1.0, facecolor='1.0', alpha=1.0, ls='dotted')
>>> plt.axhline(y=0.5, ls='dotted', color='k')
>>> plt.yticks([0.0, 0.5, 1.0])
>>> plt.ylim(-0.1, 1.1)
```

```
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi (z)$')
>>> plt.show()
```

Come risultato dell'esecuzione dell'esempio di codice precedente, vedremo una curva a "S" (sigmoid), (Figura 3.2).

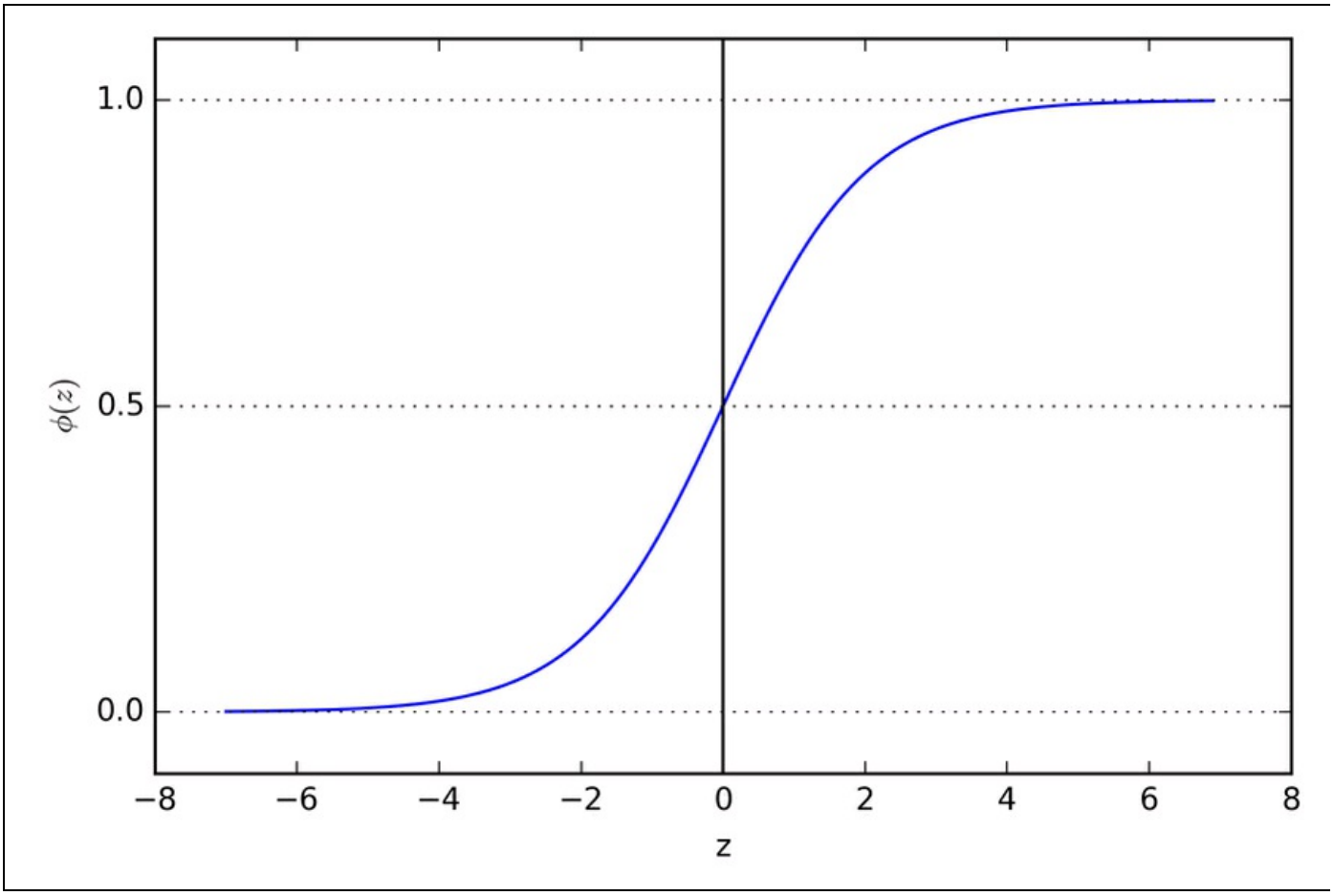


Figura 3.2

Possiamo vedere che $\phi(z)$ si avvicina a 1 se *z* tende all'infinito ($z \to \infty$), in quanto e^{-z} diviene molto piccolo per grandi valori di *z*. Analogamente, $\phi(z)$ si avvicina a 0 per $z \to -\infty$, a causa di un denominatore sempre più grande. Pertanto, concludiamo che questa funzione sigmoid prende in input dei numeri reali e li trasforma in valori compresi nell'intervallo [0, 1], con un'intercettazione a $\phi(z) = 0.5$.

Per costruire gli elementi intuitivi del modello a regressione logistica, possiamo correlarlo con la nostra precedente implementazione di Adaline tratta dal Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*. In Adaline, abbiamo utilizzato come funzione di attivazione la funzione identità $\phi(z) = z$. Nella regressione logistica, questa funzione di attivazione diviene semplicemente la

funzione sigmoid che abbiamo definito in precedenza, come illustrato nella Figura 3.3.

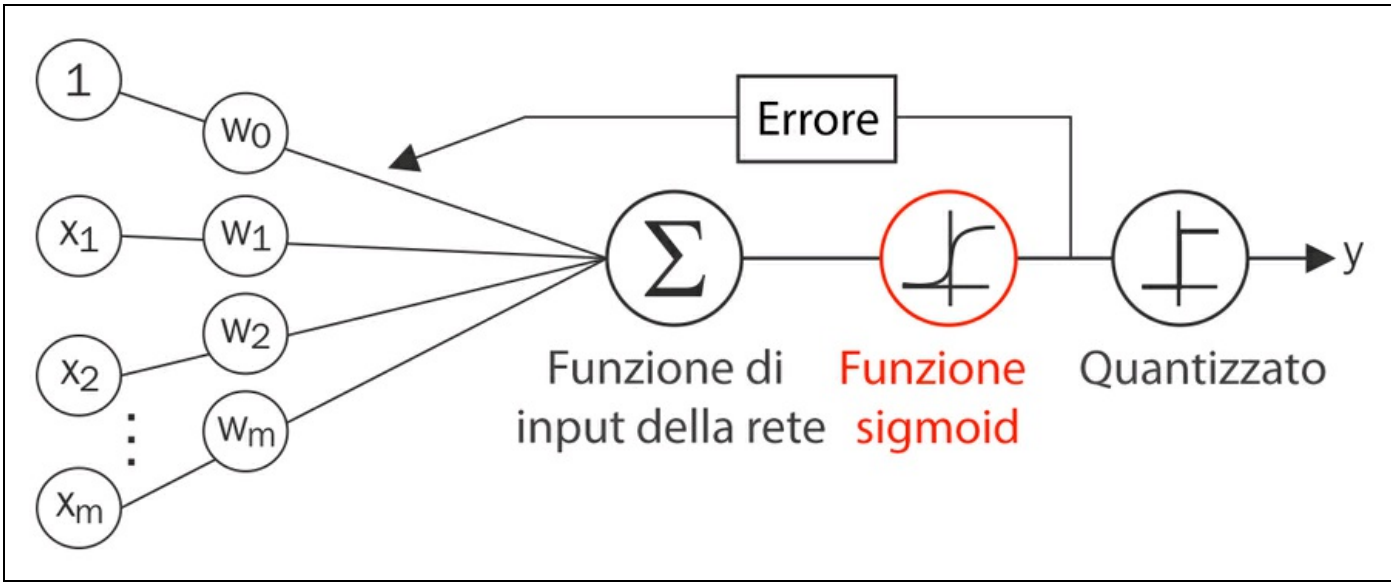


Figura 3.3

L'output della funzione sigmoid viene interpretato come la probabilità che uno specifico campione appartenga alla classe 1

$$\phi(z) = P(y = 1 \mid \boldsymbol{x}; \boldsymbol{w})$$

data la sua caratteristica *x*, parametrizzata dai pesi *w*. Per esempio, se calcoliamo

$$\phi(z) = 0.8$$

per un determinato campione di un fiore, significa che la probabilità che questo campione sia un fiore Iris-Versicolor è dell'80%. Analogamente, la probabilità che questo fiore sia un fiore Iris-Setosa, può essere calcolata come

$$P(y = 0 \mid \boldsymbol{x}; \boldsymbol{w}) = 1 - P(y = 1 \mid \boldsymbol{x}; \boldsymbol{w}) = 0.2$$

ovvero il 20%. La probabilità prevista può quindi essere semplicemente convertita in un risultato binario tramite un quantizzatore (funzione di passo unitario):

$$\hat{y} = \begin{cases} 1 & if\ \phi(z) \geq 0.5 \\ 0 & altrimenti \end{cases}$$

Osservando il grafico sigmoid precedente, questo è equivalente a:

$$\hat{y} = \begin{cases} 1 & if\ z \geq 0.0 \\ 0 & altrimenti \end{cases}$$

In pratica, vi sono molte applicazioni in cui non siamo interessati tanto alle etichette delle classi che vengono previste, ma intendiamo soprattutto stimare la probabilità di appartenenza a una classe. La regressione logistica viene utilizzata nelle previsioni meteorologiche, per esempio, non solo per prevedere se pioverà in un determinato giorno, ma anche per indicare le probabilità di pioggia. Analogamente, la regressione logistica può essere utilizzata per prevedere le probabilità che un paziente abbia una determinata malattia sulla base dei suoi sintomi. Questo è il motivo per cui la regressione logistica gode di ampia popolarità nel campo della medicina.

I pesi della funzione di costo logistico

Abbiamo visto come si possa utilizzare il modello a regressione logistica per prevedere le probabilità e le etichette delle classi. Ora parleremo brevemente dei parametri del modello, per esempio del peso *w*. Nel capitolo precedente, abbiamo definito la funzione di costi SSE (Sum Squared Error):

$$J(\boldsymbol{w}) = \sum_i \frac{1}{2}\left(\phi\left(z^{(i)}\right) - y^{(i)}\right)^2$$

L'abbiamo minimizzata in modo da determinare i pesi *w* per il modello di classificazione Adaline. Per spiegare come possiamo derivare la funzione del costo per la regressione logistica, definiamo innanzitutto la probabilità *L* che vogliamo massimizzare quando realizziamo un modello a regressione logistica, supponendo che i singoli campioni del nostro dataset siano tutti indipendenti fra loro. La formula è la seguente:

$$L(\boldsymbol{w}) = P(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{w}) = \prod_{i=1}^{n} P\left(y^{(i)} \mid x^{(i)}; \boldsymbol{w}\right) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

In pratica, è più facile massimizzare il logaritmo naturale di questa equazione, che è chiamata funzione di log-probabilità:

$$l(\boldsymbol{w}) = \log L(\boldsymbol{w}) = \sum_{i=1}^{n} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)$$

Innanzitutto, applicando la funzione logaritmo, si riducono le possibilità di un underflow numerico, che può verificarsi se le probabilità sono molto ridotte. In secondo luogo, possiamo convertire il prodotto dei fattori in una sommatoria di fattori, che semplifica il calcolo della derivata di questa funzione tramite il trucco dell'addizione, come si può ricordare dal calcolo algebrico.

Ora potremo utilizzare un algoritmo di ottimizzazione, come quello a discesa del gradiente, per massimizzare questa funzione di log-probabilità. Alternativamente, riscriviamo la log-probabilità come una funzione di costi J che può essere minimizzata utilizzando la discesa del gradiente, come descritto nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*:

$$J(\boldsymbol{w}) = \sum_{i=1}^{n} -\log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)$$

Per comprendere meglio questa funzione di costo, diamo un'occhiata al costo che calcoliamo per un'unica istanza mono-campione:

$$J\left(\phi(z), y; \boldsymbol{w}\right) = -y \log\left(\phi(z)\right) - (1 - y) \log\left(1 - \phi(z)\right)$$

Osservando l'equazione precedente, possiamo vedere che il primo termine diviene 0 se $y = 0$ e il secondo termine diviene 0 se $y = 1$:

$$J\left(\phi(z), y; \boldsymbol{w}\right) = \begin{cases} -\log\left(\phi(z)\right) & \text{if } y = 1 \\ -\log\left(1 - \phi(z)\right) & \text{if } y = 0 \end{cases}$$

Il seguente grafico illustra il costo di classificazione di un'istanza mono-campione per vari valori di $\phi(z)$ (Figura 3.4).

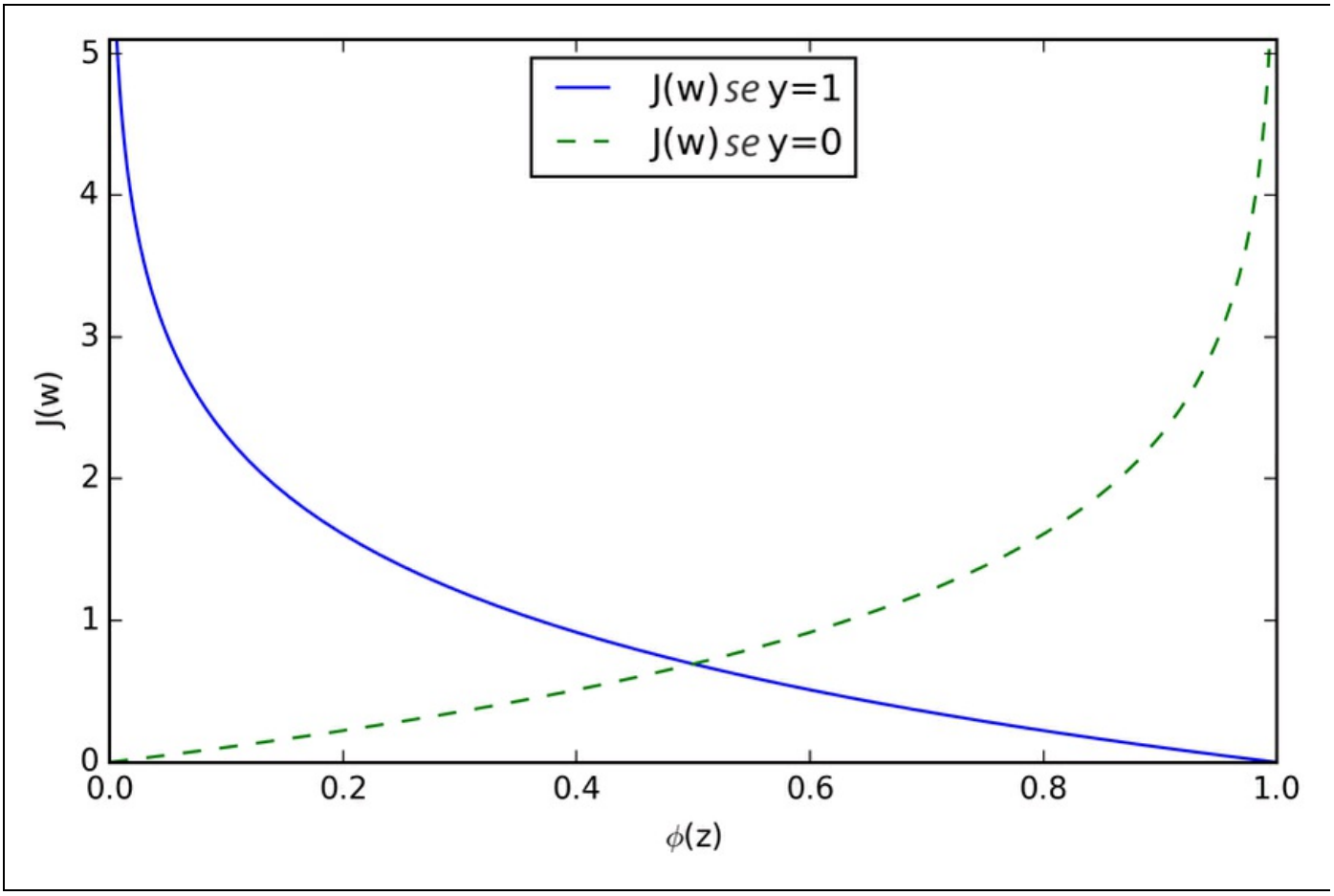


Figura 3.4

Possiamo vedere che il costo si avvicina a 0 (la linea continua) se prevediamo correttamente che un campione appartenga alla classe 1. Analogamente, possiamo vedere sull'asse *y* che il costo si avvicina a 0 se prevediamo correttamente $y = 0$ (linea tratteggiata). Tuttavia, se la previsione è errata, il costo tende all'infinito. L'idea è quella di penalizzare le previsioni errate con costi sempre più alti.

Addestramento di un modello a regressione logistica con scikit-learn

Se dovessimo implementare da soli la regressione logistica, potremmo semplicemente sostituire la funzione di costo J nella nostra implementazione Adaline del Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, impiegando la seguente nuova funzione di calcolo dei costi:

$$J(\boldsymbol{w}) = -\sum_i y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)$$

Questo calcolerebbe il costo della classificazione di tutti i campioni di addestramento per epoch e finiremmo con ottenere un modello a regressione logistica funzionante. Tuttavia, poiché scikit-learn implementa una versione molto ottimizzata della regressione logistica, che supporta anche, già da sé, le impostazioni multiclasse, salteremo l'implementazione e useremo la classe `sklearn.linear_model.LogisticRegression` e anche il noto metodo `fit` per addestrare il modello sul dataset di addestramento standardizzato dei fiori:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=1000.0, random_state=0)
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=lr,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Dopo aver adattato il modello sui dati di addestramento, abbiamo tracciato le regioni decisionali, i campioni di addestramento e i campioni di test, come si può vedere nella Figura 3.5.

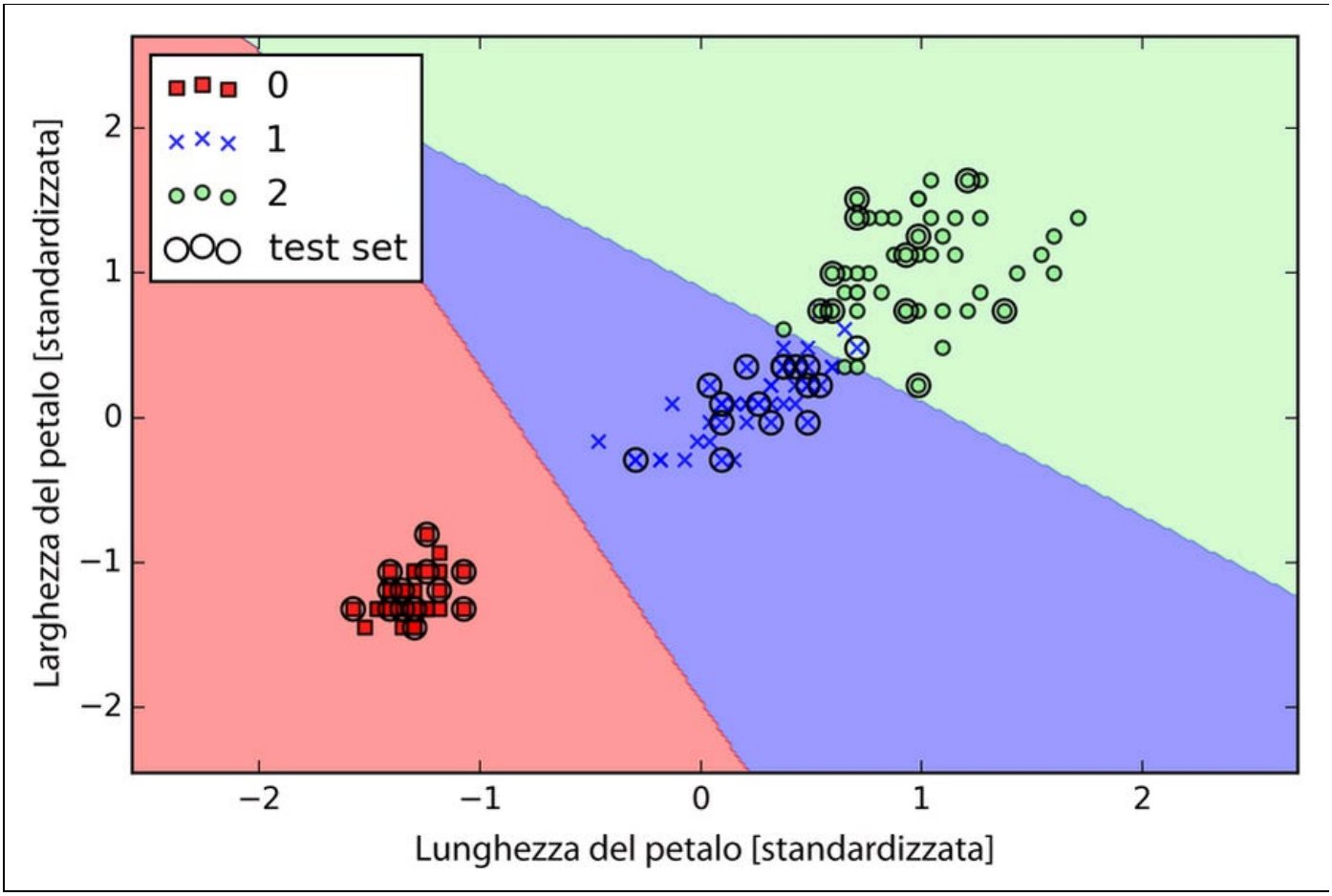


Figura 3.5

Osservando il codice precedente che abbiamo utilizzato per addestrare il modello `LogisticRegression`, potreste chiedervi quale sia il significato del "misterioso" parametro C. Ci arriveremo fra poco, ma esaminiamo brevemente i concetti di overfitting e regolarizzazione.

Inoltre possiamo prevedere la probabilità di appartenenza alla classe dei campioni tramite il metodo `predict_proba`. Per esempio, possiamo prevedere le probabilità del primo campione Iris-Setosa:

```
>>> lr.predict_proba(X_test_std[0,:])
```

Otteniamo il seguente array:

```
array([[ 0.000,  0.063,  0.937]])
```

L'array precedente ci dice che il modello prevede con una probabilità del 93.7% che il campione appartenga alla classe Iris-Virginica e con il 6.3% di probabilità che il campione sia un fiore Iris-Versicolor.

Possiamo dimostrare che l'aggiornamento dei pesi nella regressione logistica tramite la discesa del gradiente sia in effetti uguale all'equazione che abbiamo utilizzato in Adaline nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*. Iniziamo calcolando la derivata parziale della funzione di log-probabilità rispetto al peso j-esimo:

$$\frac{\partial}{\partial w_j} l(\boldsymbol{w}) = \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

Prima di procedere, calcoliamo la derivata parziale della funzione sigmoid:

$$\frac{\partial}{\partial z} \phi(z) = \frac{\partial}{\partial z} \frac{1}{1+e^{-z}} = \frac{1}{\left(1+e^{-z}\right)^2} e^{-z} = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}} \right)$$

$$= \phi(z)\left(1-\phi(z)\right)$$

Ora possiamo sostituire

$$\frac{\partial}{\partial z} \phi(z) = \phi(z)\left(1-\phi(z)\right)$$

nella nostra prima equazione, per ottenere la seguente:

$$\left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)} \right)\frac{\partial}{\partial w_j}\phi(z)$$

$$= \left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)} \right)\phi(z)\left(1-\phi(z)\right)\frac{\partial}{\partial w_j}z$$

$$= \left(y\left(1-\phi(z)\right) - (1-y)\phi(z)\right)x_j$$

$$= \left(y-\phi(z)\right)x_j$$

Ricordate che l'obiettivo consiste nel trovare i pesi che massimizzano la log-probabilità, in modo che eseguiremo l'aggiornamento per ogni peso nel seguente modo:

$$w_j := w_j + \eta\sum_{i=1}^{n}\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)x^{(i)}$$

Poiché aggiorniamo simultaneamente tutti i pesi, possiamo scrivere la regola di aggiornamento generale nel seguente modo:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}$$

Definiamo $\Delta\boldsymbol{w}$nel seguente modo:

$$\Delta\boldsymbol{w} = \eta\nabla l\left(\boldsymbol{w}\right)$$

Poiché la massimizzazione della log-probabilità è uguale alla minimizzazione della funzione di costo Jche abbiamo definito in precedenza, possiamo scrivere la regola di aggiornamento della discesa del gradiente nel seguente modo:

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = \eta\sum_{i=1}^{n}\left(y^{(i)} - \phi\left(z^{(i)}\right)x^{(i)}\right)$$

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w},\ \Delta\boldsymbol{w} = -\eta\nabla J\left(\boldsymbol{w}\right)$$

Questa equivale alla regola di discesa del gradiente di Adaline nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*.

Risolvere l'overfitting tramite la regolarizzazione

Quello dell'*overfitting* è un problema comune nelle attività di machine learning: un modello si comporta bene sui dati di addestramento, ma non è in grado di generalizzarsi altrettanto bene su nuovi dati (dati di test). Se un modello soffre di questo difetto, diciamo anche che ha un'elevata varianza, il che può essere causato dal fatto di avere troppi parametri, i quali producono un modello troppo complesso per i dati sottostanti. Analogamente, il nostro modello può anche soffrire del problema opposto, *underfitting* (high bias, elevata discrepanza): la sua limitata complessità non è in grado di catturare lo schema presente nei dati di addestramento e pertanto soffre di scarse prestazioni anche su nuovi dati.

Sebbene abbiamo solo incontrato, finora, solo modelli di classificazione lineari, il problema dell'overfitting e dell'underfitting può essere illustrato meglio utilizzando un confine decisionale complesso, non lineare, come nella Figura 3.6.

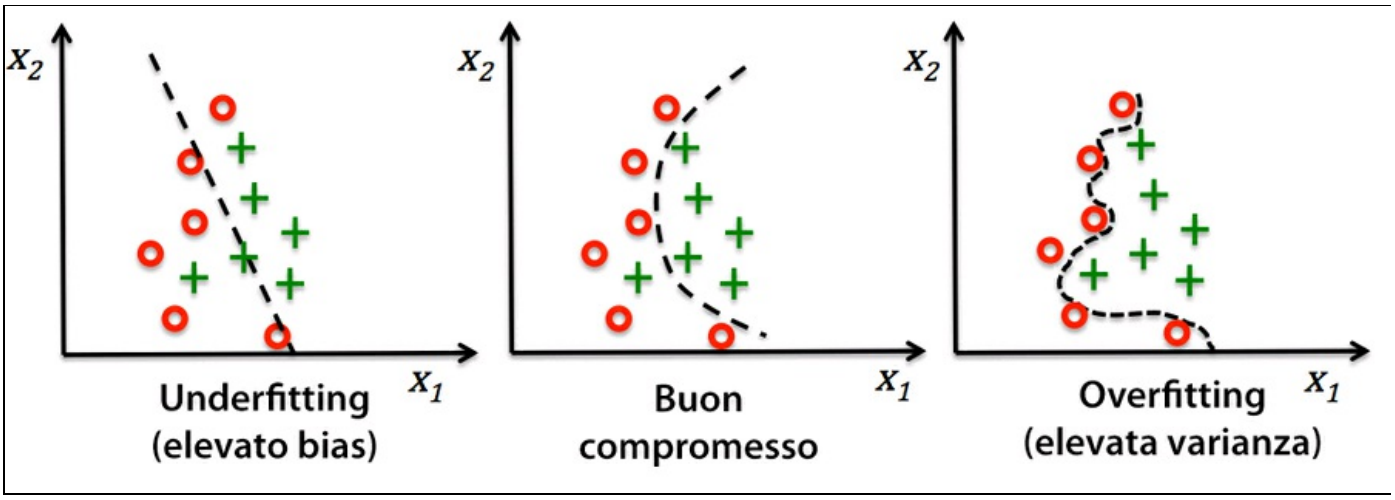


Figura 3.6

NOTA

La varianza misura la coerenza (o la variabilità) della previsione del modello per una determinata istanza del campione se ripetiamo l'addestramento del modello più volte, per esempio, su insiemi differenti del dataset di addestramento. Possiamo dire che il modello è sensibile alla casualità dei dati d'addestramento. Al contrario, il bias, la discrepanza, misura quanto distanti sono le previsioni rispetto ai valori corretti in generale, se ricostruiamo il modello più volte su dataset di addestramento differenti; il bias misura l'errore sistematico che non è legato alla casualità.

Un modo per trovare un buon compromesso fra bias e varianza consiste nell'ottimizzare la complessità del modello tramite la regolarizzazione. La regolarizzazione è un metodo molto utile per gestire la colinearità (elevata correlazione fra le caratteristiche), per eliminare il rumore dai dati e per, alla fine,

prevenire il problema dell'overfitting. Il concetto su cui si basa la regolarizzazione consiste nell'introdurre delle informazioni aggiuntive (bias) per penalizzare i pesi estremi del parametro. La forma più comune di regolarizzazione è chiamata *regolarizzazione L2* (chiamata anche riduzione L2 o decadimento dei pesi) che può essere scritta nel seguente modo:

$$\frac{\lambda}{2}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

Qui, λ è il cosiddetto parametro di regolarizzazione.

NOTA

La regolarizzazione è un altro motivo per cui l'operazione di riduzione in scala delle caratteristiche, come la standardizzazione, è importante. Perché la regolarizzazione funzioni correttamente, dobbiamo assicurarci che tutte le caratteristiche operino su una scala analoga.

Per applicare la regolarizzazione, abbiamo solo bisogno di aggiungere il termine di regolarizzazione alla funzione di costo che abbiamo definito per la regressione logistica, in modo da comprimere i pesi.

$$J(\boldsymbol{w}) = \left[\sum_{i=1}^{n}\left(-\log\left(\phi\left(z^{(i)}\right)\right) + \left(1-y^{(i)}\right)\left(-\log\left(1-\phi(z)\right)\right)\right)\right] + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

Tramite il parametro di regolarizzazione λ, possiamo quindi controllare l'adattamento ai dati di addestramento contenendo l'intensità dei pesi. Incrementando il valore di λ, incrementiamo l'intensità della regolarizzazione.

Il parametro `C` che è implementato per la classe `LogisticRegression` in scikit-learn proviene da una convenzione presente nelle macchine a vettori di supporto, argomento del prossimo paragrafo. `C` è direttamente correlato col parametro di regolarizzazione λ, in quanto è il suo inverso:

$$C = \frac{1}{\lambda}$$

Pertanto possiamo riscrivere la funzione di costo regolarizzata della regressione logistica nel seguente modo:

$$J(\boldsymbol{w}) = C\left[\sum_{i=1}^{n}\left(-\log\left(\phi\left(z^{(i)}\right)\right) + \left(1-y^{(i)}\right)\right)\left(-\log\left(1-\phi(z)\right)\right)\right] + \frac{1}{2}\|\boldsymbol{w}\|^2$$

Di conseguenza, decrementando il valore del parametro di regolarizzazione inversa c incrementiamo l'intensità della regolarizzazione, che possiamo rappresentare tracciando il percorso di regolarizzazione L2 per i due coefficienti di peso:

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10**c, random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...          label='petal length')
>>> plt.plot(params, weights[:, 1], linestyle='--',
...          label='petal width')
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

Tramite il codice precedente, abbiamo previsto dieci modelli a regressione logistica con valori diversi del parametro di regolarizzazione inversa c. Per gli scopi dell'illustrazione (Figura 3.7), abbiamo raccolto solo i coefficienti di peso della classe 2 rispetto a tutti i classificatori. Ricordate che utilizziamo la tecnica OvR per la classificazione multiclasse.

Come possiamo vedere nel grafico risultante, i coefficienti di peso si riducono se riduciamo il parametro c, ovvero se incrementiamo l'intensità della regolarizzazione.

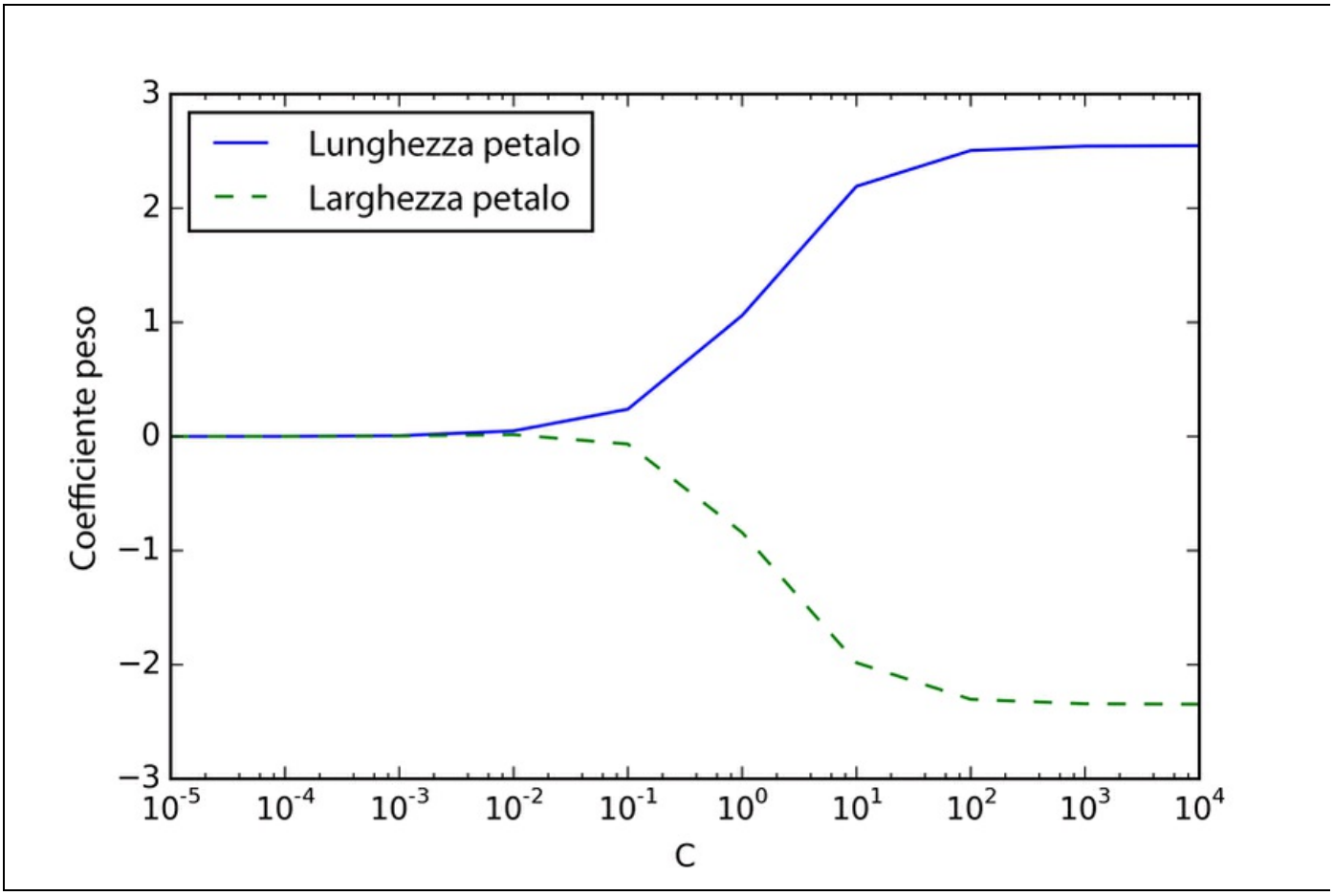


Figura 3.7

NOTA

Poiché una trattazione approfondita dei singoli algoritmi di classificazione non rientra negli scopi di questo libro, a tutti coloro che vogliono conoscere qualcosa di più sulla regressione logistica è consigliabile la lettura del testo del Dr. Scott Menard, *Logistic Regression: From Introductory to Advanced Concepts and Applications*, Sage Publications.

Classificazione a massimo margine con le macchine a vettori di supporto

Un altro algoritmo molto potente e ampiamente utilizzato è la macchina a vettori di supporto (*support vector machine – SVM*), che può essere considerata come un'estensione del perceptron. Utilizzando l'algoritmo perceptron, abbiamo minimizzato gli errori di mancata classificazione. Tuttavia, con SVM, il nostro obiettivo di ottimizzazione consiste nel massimizzare il margine. Il margine è definito come la distanza fra l'iperpiano di separazione (confine decisionale) e i campioni di addestramento che sono più vicini a questo iperpiano, che sono i cosiddetti vettori di supporto. Il tutto è illustrato dalla Figura 3.8.

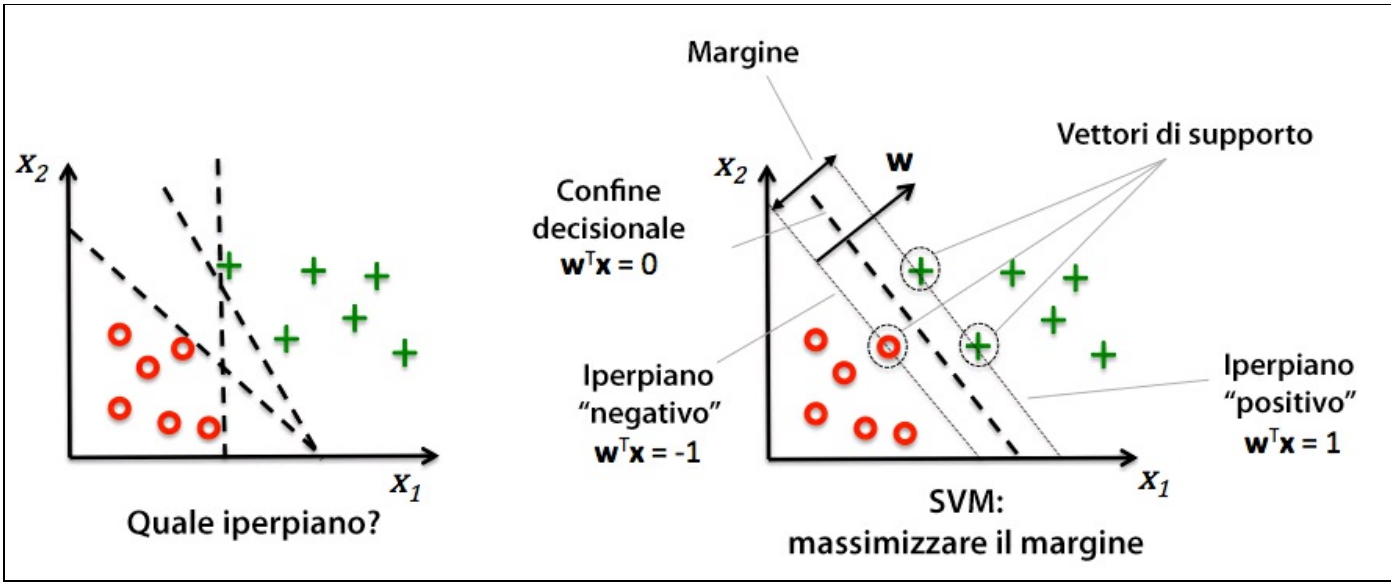


Figura 3.8

Individuazione del massimo margine

La logica che porta ad avere confini decisionali con ampi margini è il fatto che questi tendono ad avere un errore di generalizzazione più basso, mentre i modelli con margini più ridotti sono più soggetti a overfitting. Per capire meglio l'idea della massimizzazione del margine, diamo un'occhiata agli iperpiani positivo e negativo che sono paralleli al confine decisionale e che possono essere descritti nel seguente modo:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{pos} = 1 \qquad (1)$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{neg} = -1 \quad (2)$$

Se sottraiamo queste due equazioni lineari l'una dall'altra, otteniamo:

$$\Rightarrow \boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg} \right) = 2$$

Possiamo normalizzare il tutto per la lunghezza del vettore *w*, che è definito nel seguente modo:

$$\|\boldsymbol{w}\| = \sqrt{\sum_{j=1}^{m} w_j^2}$$

Pertanto arriviamo alla seguente equazione:

$$\frac{\boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg} \right)}{\|\boldsymbol{w}\|} = \frac{2}{\|\boldsymbol{w}\|}$$

Il lato sinistro dell'equazione precedente può essere interpretato come la distanza fra l'iperpiano negativo e quello positivo, ovvero il cosiddetto margine che vogliamo massimizzare.

Ora la funzione obiettivo di SVM diviene la massimizzazione di questo margine massimizzando $\frac{2}{\|\boldsymbol{w}\|}$ sotto il vincolo che i campioni siano classificati correttamente, il che può essere scritto nel seguente modo:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 \; if \; y^{(i)} = 1$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} < -1 \; if \; y^{(i)} = -1$$

Queste due equazioni dicono sostanzialmente che tutti i campioni negativi devono rientrare su un lato dell'iperpiano negativo, mentre tutti i campioni positivi devono rientrare sull'altro lato dell'iperpiano positivo. Questo può essere scritto in forma più compatta nel seguente modo:

$$y^{(i)}\left(w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)}\right) \geq 1 \ \forall_i$$

In pratica, però, è più facile minimizzare il termine reciproco $\frac{1}{2}\|\boldsymbol{w}\|^2$, che può essere risolto tramite la programmazione quadratica. Tuttavia, una discussione dettagliata sulla programmazione quadratica non rientra negli scopi di questo libro, ma se siete interessati, potete scoprire di più sulle macchine a vettori di supporto in Vladimir Vapnik, *The Nature of Statistical Learning Theory*, Springer Science & Business Media o nell'eccellente spiegazione di Chris J.C. Burges, *A Tutorial on Support Vector Machines for Pattern Recognition* ("Data mining and knowledge discovery", 2(2):121–167, 1998).

Il caso separabile non linearmente utilizzando variabili slack

Sebbene non vogliamo approfondire troppo i concetti matematici su cui si basa la classificazione a margine, menzioniamo brevemente la variabile slack ξ. È stata introdotta da Vladimir Vapnik nel 1995 e ha condotto alla cosiddetta classificazione a margine soft. La motivazione per l'introduzione della variabile slack ξ è il fatto che i vincoli di linearità devono essere attenuati per i dati separabili in modo non lineare, in modo da consentire la convergenza dell'ottimizzazione in presenza di errate classificazioni sotto l'appropriata penalizzazione dei costi.

La variabile slack per valori positivi viene semplicemente aggiunta ai vincoli lineari:

$$\boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 \ if \ \ y^{(i)} = 1 - \xi^{(i)}$$

$$\boldsymbol{w}^T \boldsymbol{x}^{(i)} < -1 \ if \ \ y^{(i)} = 1 + \xi^{(i)}$$

Pertanto il nuovo obiettivo da minimizzare (soggetto ai vincoli precedenti) diviene:

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

Utilizzando la variabile c, possiamo quindi controllare la penalizzazione per un'errata classificazione. Valori ampi di c corrispondono a grandi penalizzazioni di un errore, mentre se scegliamo valori più ridotti di c siamo meno rigidi nei confronti degli errori di errata classificazione. Possiamo quindi utilizzare il parametro c per controllare l'ampiezza del margine e pertanto ottimizzare il compromesso fra bias e varianza, come illustrato dalla Figura 3.9.

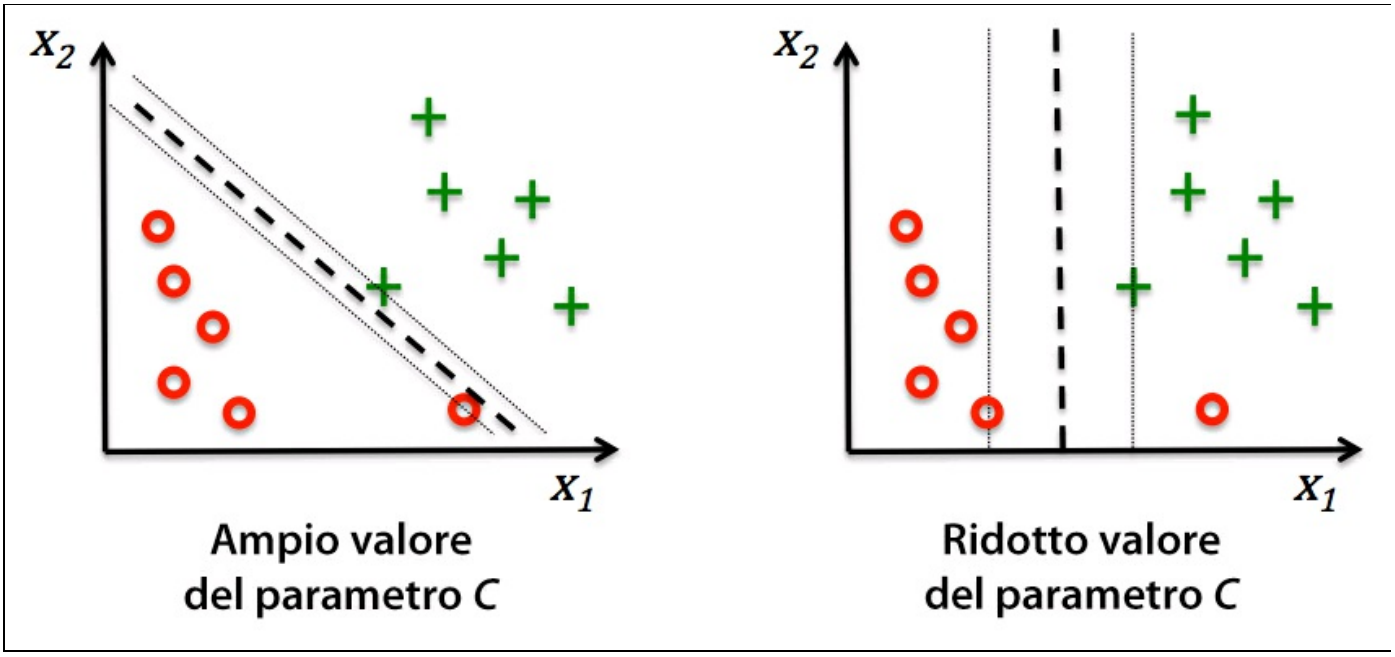


Figura 3.9

Questo concetto è correlato alla regolarizzazione, di cui abbiamo trattato precedentemente nel contesto della regressione regolarizzata, dove incrementando il valore di c si aumenta il bias e si riduce la varianza del modello.

Ora che abbiamo trattato i concetti di base riguardanti l'algoritmo SVM lineare, addestriamo un modello SVM per classificare i diversi fiori del nostro dataset Iris:

```
>>> from sklearn.svm import SVC
>>> svm = SVC(kernel='linear', C=1.0, random_state=0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Le regioni decisionali del SVM visualizzate dopo aver eseguito il codice precedente sono rappresentate nella Figura 3.10.

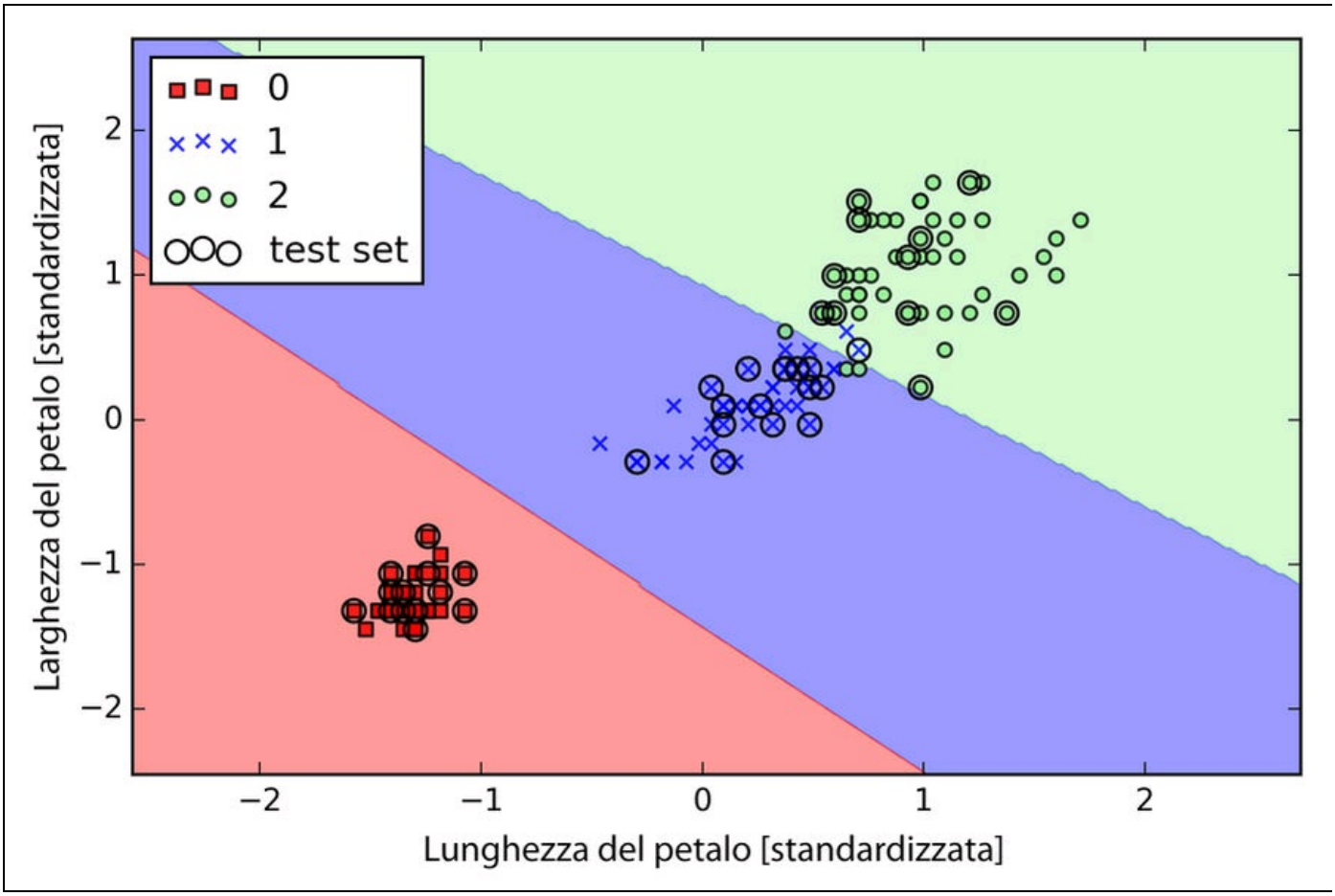


Figura 3.10

NOTA

Nelle attività pratiche di classificazione, la regressione logistica lineare e una SVM lineare spesso forniscono risultati simili. La regressione logistica tende a massimizzare le probabilità condizionali dei dati di addestramento, il che rende il tutto più soggetto alle anomalie rispetto alle SVM. Le SVM si preoccupano soprattutto dei punti che sono più vicini al confine decisionale (vettori di supporto). D'altra parte, la regressione logistica presenta il vantaggio di essere un modello più semplice, che può essere implementato con maggiore facilità. Inoltre, i modelli a regressione logistica possono essere aggiornati con facilità, il che è interessante quando si lavora con dati in streaming.

Implementazioni alternative in scikit-learn

Le classi `Perceptron` e `LogisticRegression` che abbiamo utilizzato nei paragrafi precedenti tramite scikit-learn fanno uso della libreria LIBLINEAR, una libreria C/C++ a elevata ottimizzazione sviluppata presso la National Taiwan University (http://www.csie.ntu.edu.tw/~cjlin/liblinear/). Analogamente, la classe SVC che abbiamo utilizzato per

addestrare una SVM fa uso della libreria LIBSVM, che è una libreria C/C++ equivalente, specializzata in SVM (http://www.csie.ntu.edu.tw/~cjlin/libsvm/).

Il vantaggio di utilizzare LIBLINEAR e LIBSVM rispetto alle implementazioni native in Python è il fatto che esse consentono di eseguire un rapidissimo addestramento di una grande quantità di classificatori lineari. Tuttavia, talvolta i nostri dataset sono troppo estesi per poter rientrare nella memoria del computer. Pertanto, scikit-learn offre anche delle implementazioni alternative tramite la classe `SGDClassifier`, che supporta anche l'apprendimento online tramite il metodo `partial_fit`. Il concetto su cui si basa la classe `SGDClassifier` è simile all'algoritmo a gradiente stocastico che abbiamo implementato nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, per Adaline. Potremmo inizializzare la versione del perceptron, della regressione logistica e della macchina a vettori di supporto con la versione a discesa del gradiente stocastica con i parametri standard nel seguente modo:

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
>>> svm = SGDClassifier(loss='hinge')
```

Soluzione di problemi non lineari utilizzando una SVM kernel

Un altro motivo per cui le macchine a vettori di supporto (SVM) godono di grande popolarità fra gli esperti di machine learning è il fatto che possono essere *kernel*-izzate per risolvere problemi di classificazione non lineari. Prima di discutere i concetti su cui si basa una SVM kernel, definiamo e creiamo un dataset di campioni per vedere quale possa essere l'aspetto di un problema di classificazione non lineare.

Utilizzando il codice seguente, creeremo un semplice dataset che ha l'aspetto di uno XOR-gate utilizzando la funzione `logical_xor` di NumPy, dove a 100 campioni verrà assegnata l'etichetta della classe 1 e ad altri 100 verrà assegnata l'etichetta della classe -1:

```
>>> np.random.seed(0)
>>> X_xor = np.random.randn(200, 2)
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0, X_xor[:, 1] > 0)
>>> y_xor = np.where(y_xor, 1, -1)
>>> plt.scatter(X_xor[y_xor==1, 0], X_xor[y_xor==1, 1],
...            c='b', marker='x', label='1')
>>> plt.scatter(X_xor[y_xor==-1, 0], X_xor[y_xor==-1, 1],
...            c='r', marker='s', label='-1')
>>> plt.ylim(-3.0)
>>> plt.legend()
>>> plt.show()
```

Dopo aver eseguito il codice, avremo un dataset XOR contenente del rumore casuale, come si può vedere nella Figura 3.11.

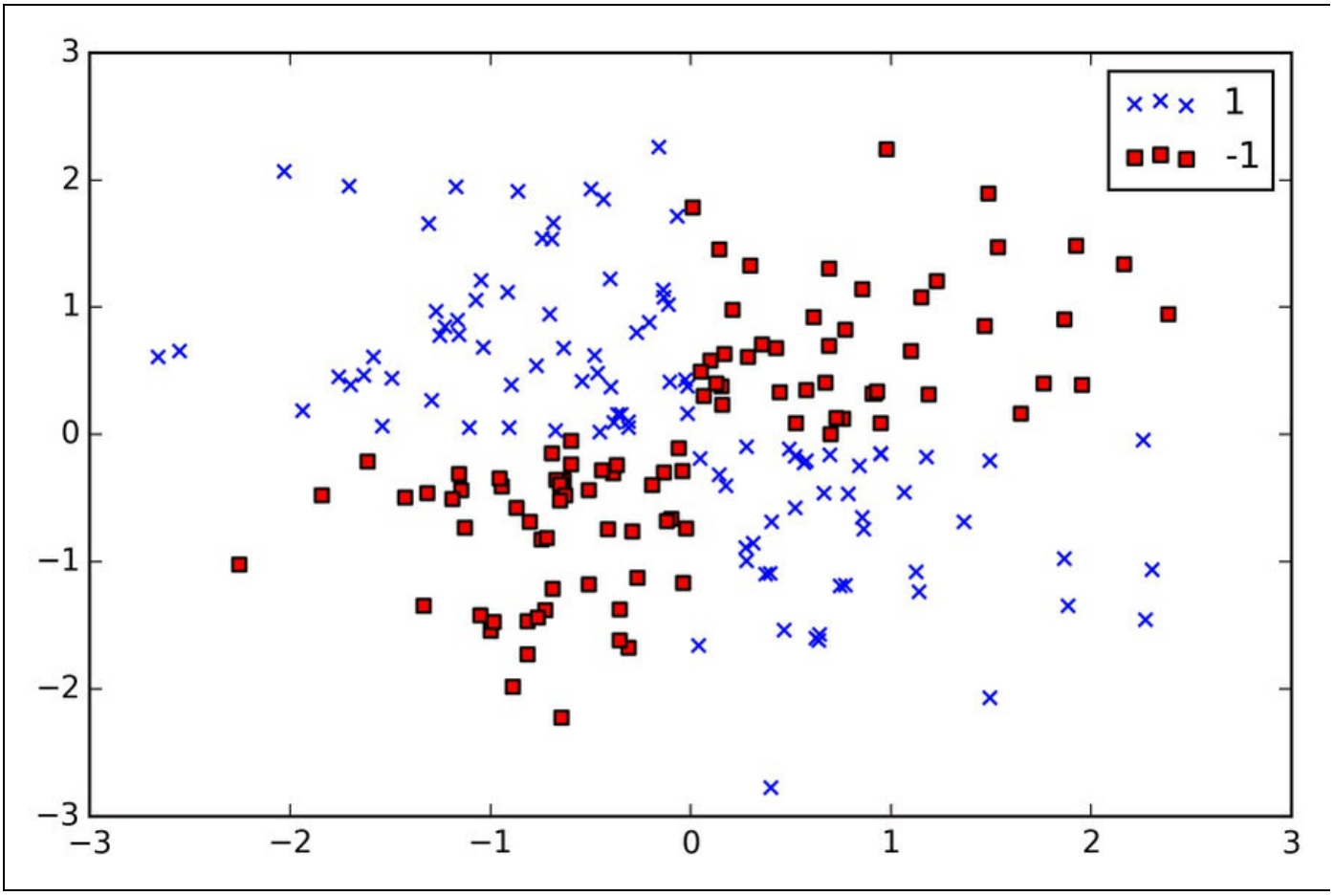


Figura 3.11

Ovviamente, non saremmo assolutamente in grado di separare i campioni della classe positiva e negativa utilizzando un iperpiano lineare come confine decisionale o i modelli a regressione logistica o a SVM lineare di cui abbiamo parlato nei paragrafi precedenti.

L'idea su cui si basano i metodi kernel per gestire dati che non sono separabili linearmente consiste nel creare combinazioni non lineari delle caratteristiche originali per proiettarle in uno spazio con maggiori dimensioni tramite una funzione di mappaggio $\phi(\cdot)$, dove tali dati divengono separabili linearmente. Come si può vedere nella Figura 3.12, possiamo trasformare un dataset bidimensionale in un nuovo spazio di caratteristiche tridimensionale dove le classi divengono separabili tramite la seguente proiezione:

$$\phi\left(x_1, x_2\right) = \left(z_1, z_2, z_3\right) = \left(x_1, x_2, x_1^2 + x_2^2\right)$$

Questo ci consente di separare le due classi rappresentate nel grafico tramite un iperpiano lineare, il quale diviene un confine decisionale non lineare se proiettato sullo spazio originario delle caratteristiche (Figura 3.12).

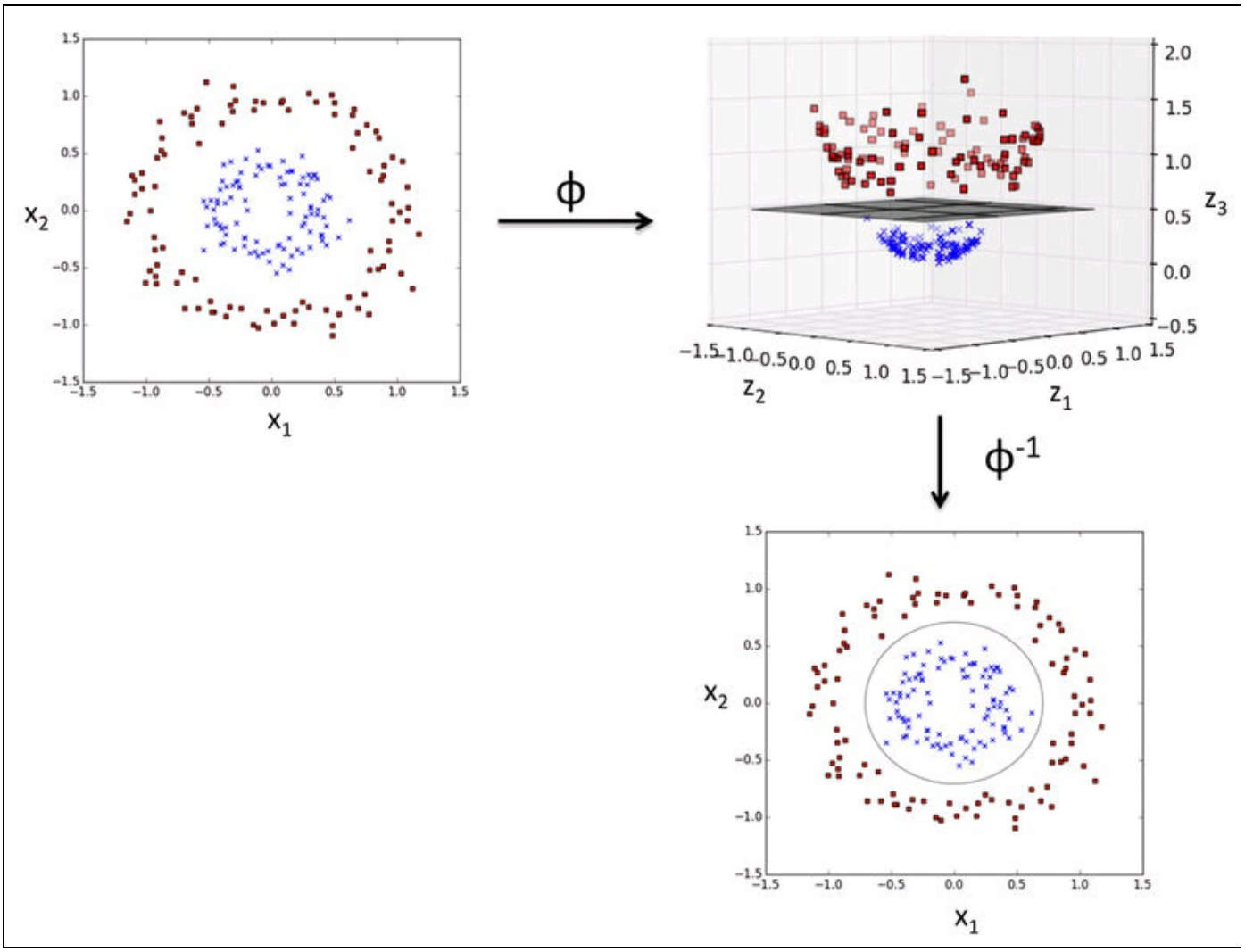


Figura 3.12

Uso della tecnica del kernel per trovare iperpiani di separazione in uno spazio di maggiori dimensioni

Per risolvere un problema non lineare utilizzando una SVM, trasformeremo i dati di addestramento proiettandoli su uno spazio di caratteristiche con dimensioni più elevate tramite una funzione di mappaggio $\phi(\cdot)$ e addestreremo un modello a SVM lineare per classificare i dati in questo nuovo spazio di caratteristiche. Poi possiamo utilizzare la stessa funzione di mappaggio $\phi(\cdot)$ per trasformare nuovi dati, ancora inutilizzati, per classificarli utilizzando il modello a SVM lineare.

Tuttavia, un problema di questo approccio al mappaggio è il fatto che la costruzione delle nuove caratteristiche è molto costosa dal punto di vista computazionale, specialmente se abbiamo a che fare con dati a elevata dimensionalità. Questo è il motivo per cui la tecnica kernel può essere utile. Anche

se non entreremo troppo nei dettagli sulla soluzione del compito di programmazione pratica per addestrare una SVM, in pratica ciò di cui abbiamo bisogno è sostituire il prodotto $\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}$ con

$$\phi\left(\boldsymbol{x}^{(i)}\right)^{T}\phi\left(\boldsymbol{x}^{(j)}\right)$$

Per risparmiare il costoso passo del calcolo esplicito di questo prodotto fra due punti, definiamo una cosiddetta *funzione kernel*:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)_{=}\phi\left(\boldsymbol{x}^{(i)}\right)^{T}\phi\left(\boldsymbol{x}^{(j)}\right)$$

Una delle più utilizzate è la funzione kernel *Radial Basis Function* (*kernel RBF*) o kernel gaussiana:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}}{2\sigma^{2}}\right)$$

Spesso questo viene semplificato in:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\gamma\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}\right)$$

Qui, $\gamma=\frac{1}{2\sigma^{2}}$ è un parametro libero che deve essere ottimizzato.

In estrema sintesi, il termine *kernel* può essere interpretato come *funzione di similarità* fra una copia di campioni. Il segno meno inverte la misura della distanza in un punteggio di similarità e, a causa del termine esponenziale, il punteggio di similarità risultante rientrerà sempre in un intervallo compreso fra 1 (per campioni esattamente simili) e 0 (per campioni molto dissimili).

Ora che abbiamo trattato a grandi linee la tecnica kernel, vediamo se possiamo addestrare una SVM kernel che sia in grado di tracciare un confine decisionale non lineare che separi bene i dati XOR. Qui utilizziamo semplicemente la classe svc di

scikit-learn che abbiamo importato in precedenza e sostituiamo il parametro kernel='linear' con kernel='rbf':

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.10, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Come possiamo vedere nel grafico risultante (Figura 3.13), l'algoritmo SVM kernel separa relativamente bene i dati XOR.

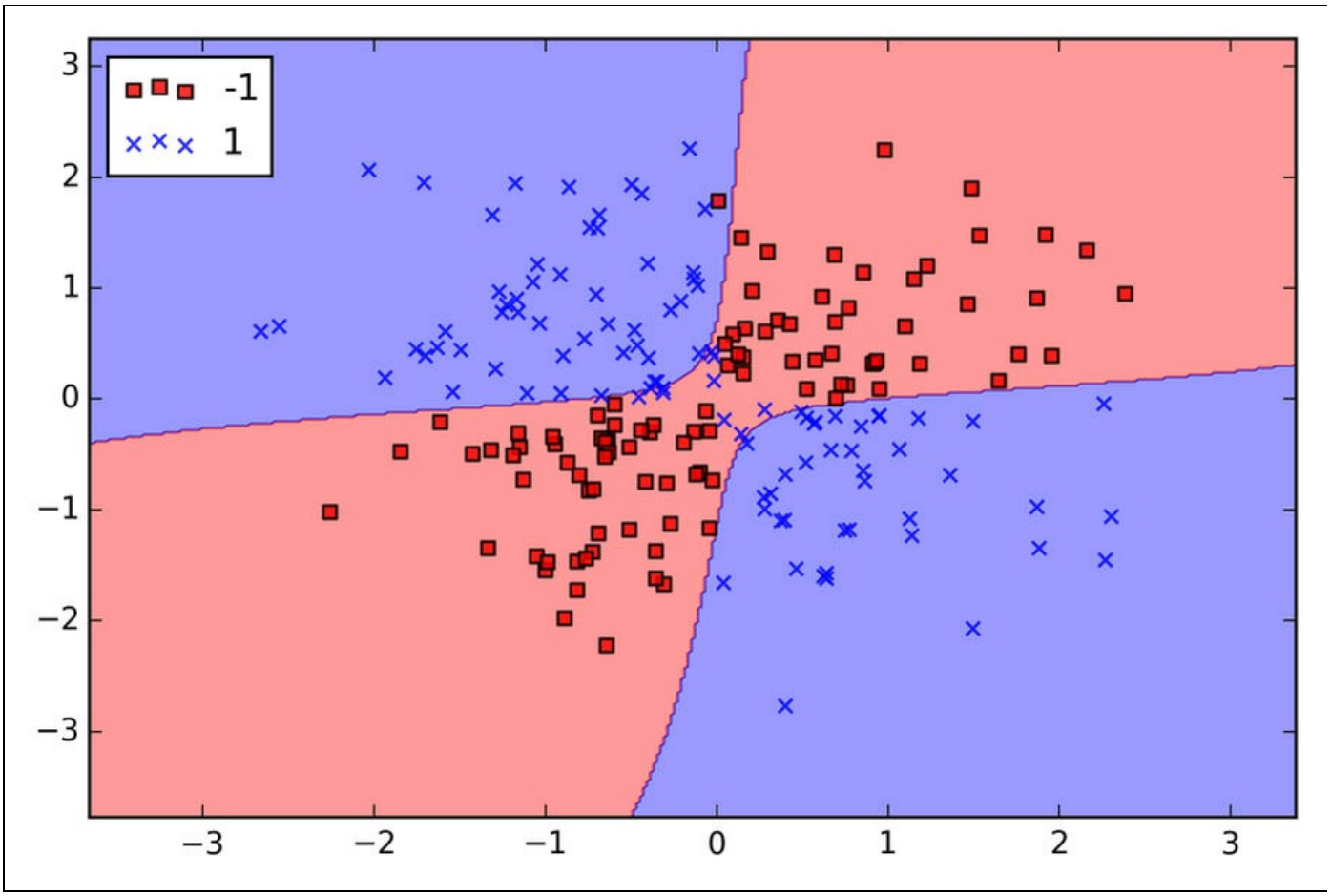


Figura 3.13

Il parametro γ, che impostiamo in gamma=0.1, può essere considerato un parametro di *cut-off* per la sfera gaussiana. Se incrementiamo il valore di γ, incrementiamo anche l'influenza o l'ampiezza dei campioni di addestramento, il che genera un confine decisionale più morbido. Per comprendere meglio il funzionamento di γ, applichiamo l'algoritmo SVM a kernel RBF al nostro dataset di fiori Iris:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Poiché scegliamo un valore relativamente ridotto per γ, il confine decisionale risultante del modello RBF una SVM kernel sarà relativamente morbido (Figura 3.14).

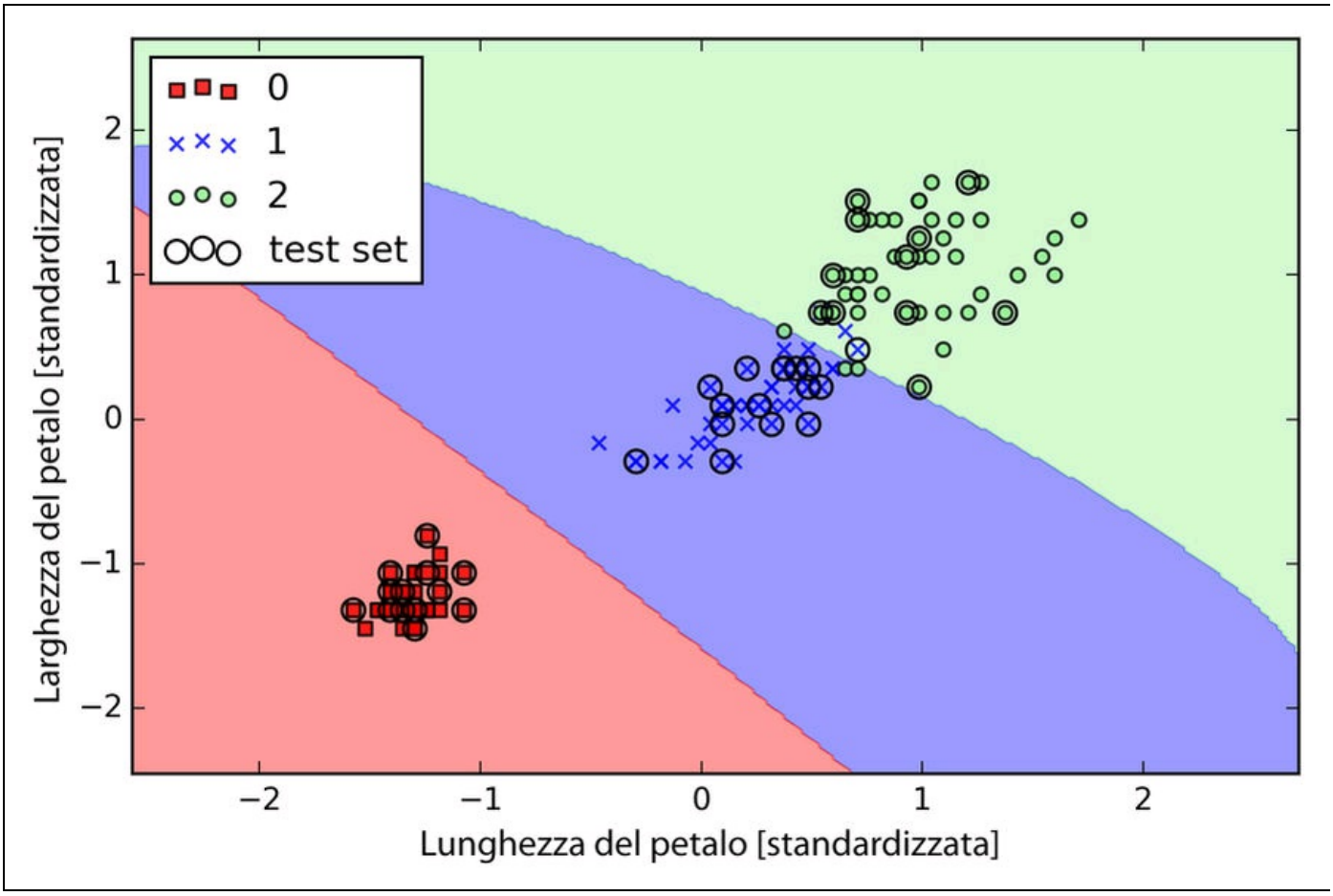


Figura 3.14

Ora incrementiamo il valore di γ e osserviamo l'effetto che si ha sul confine decisionale:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Nel grafico risultante (Figura 3.15), possiamo vedere che il confine decisionale attorno alle classi 0 e 1 è molto più stretto, utilizzando un valore relativamente ampio di γ.

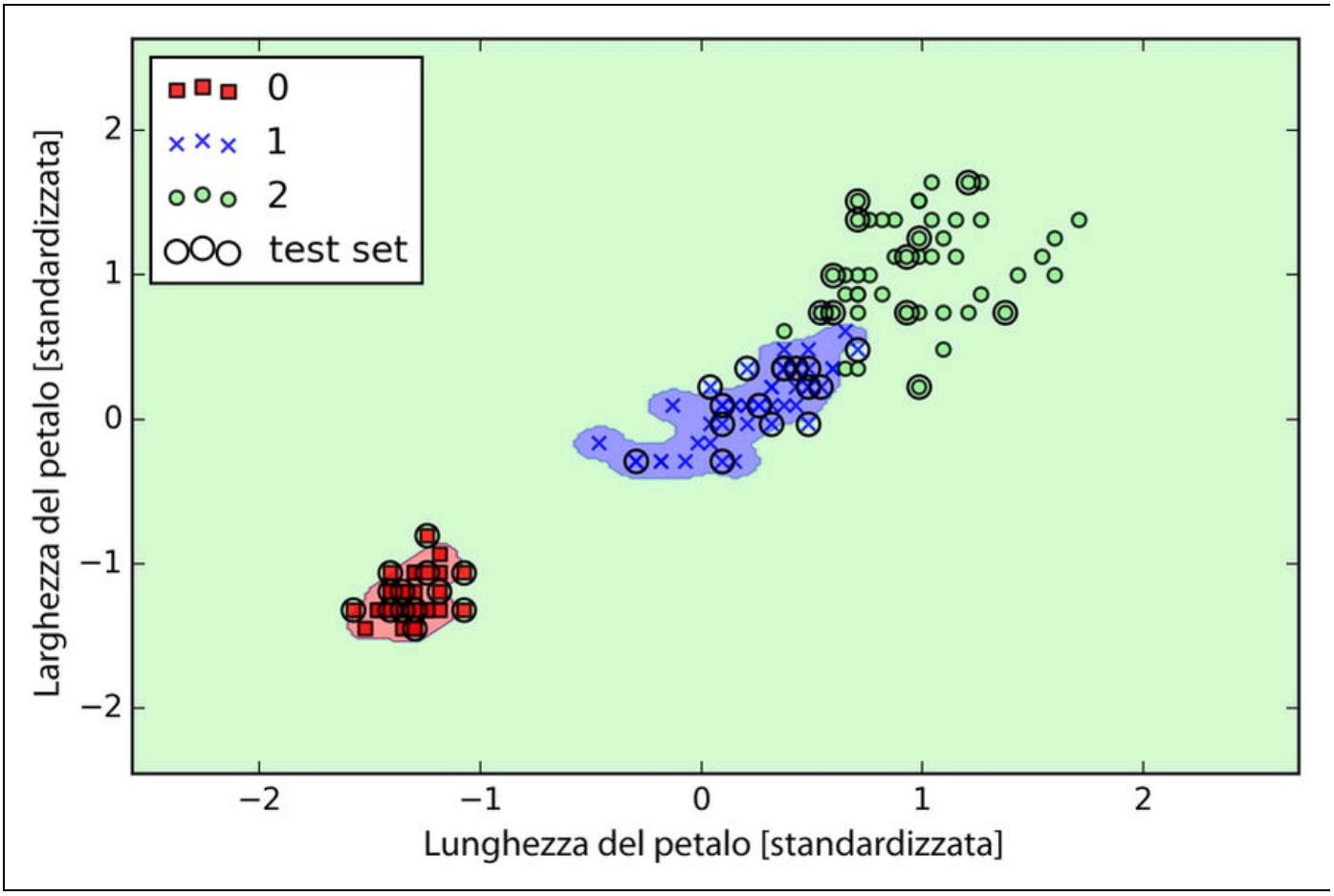


Figura 3.15

Sebbene il modello si adatti molto bene al dataset di addestramento, un classificatore di questo tipo avrà con ogni probabilità un elevato errore di generalizzazione sui dati in arrivo; questo illustra il fatto che l'ottimizzazione di γ gioca anche un ruolo importante nel controllo del disturbo di overfitting.

Apprendimento ad albero decisionale

I classificatori ad albero decisionale sono modelli interessanti se quello che ci interessa è soprattutto l'interoperabilità. Come suggerisce il nome, un modello ad albero decisionale suddivide i dati prendendo decisioni sulla base delle risposte a una serie di domande.

Consideriamo l'esempio rappresentato nella Figura 3.16, dove utilizziamo un albero decisionale per decidere l'attività da svolgere in una determinata giornata.

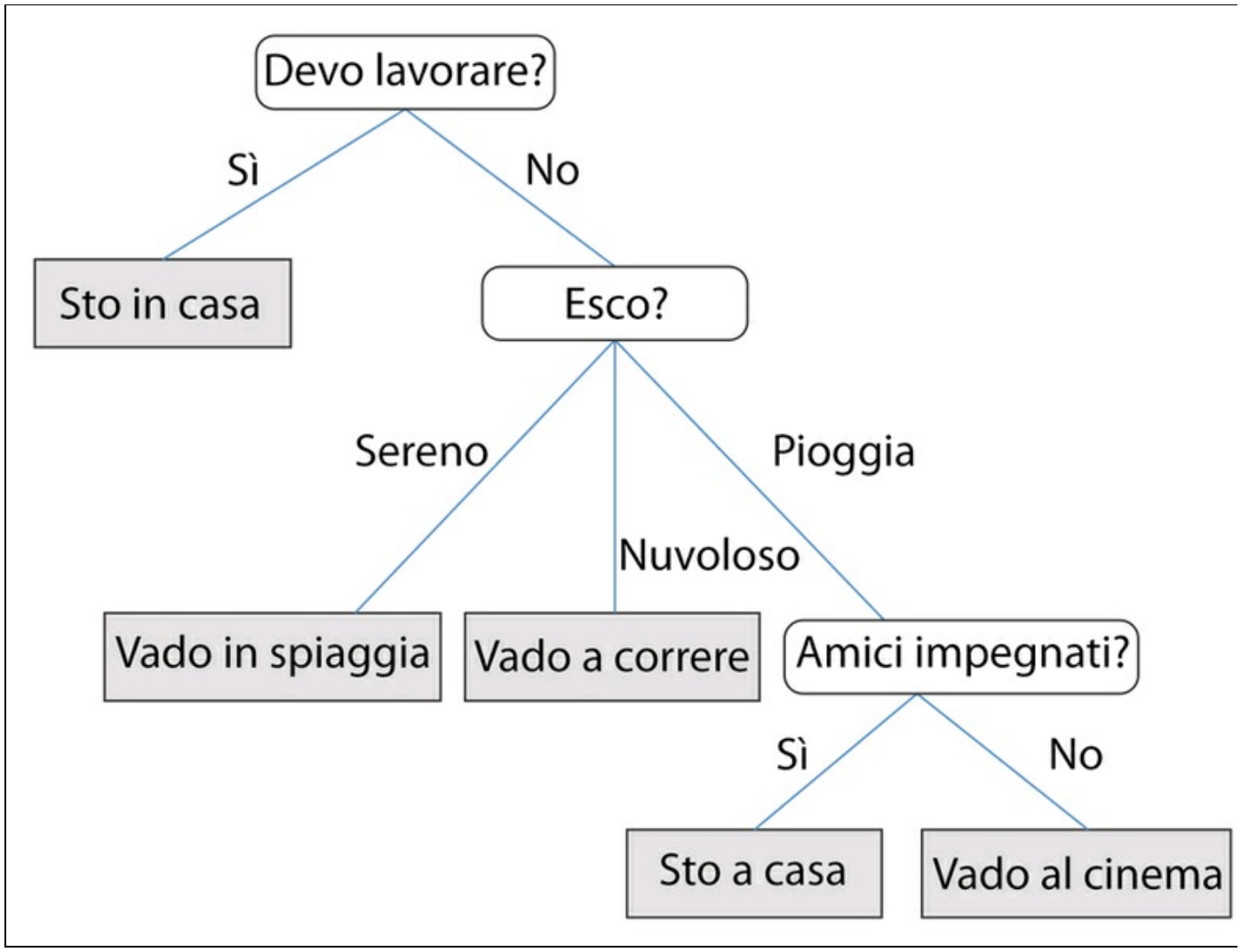


Figura 3.16

Sulla base delle caratteristiche del dataset di addestramento, il modello ad albero decisionale impara una serie di domande per determinare (tramite *inferenza*) le etichette delle classi dei campioni. Sebbene la Figura 3.16 abbia illustrato il concetto di un albero decisionale basato su variabili categoriche, lo stesso concetto si applica se le caratteristiche sono numeriche, come nel caso del dataset Iris. Per

esempio, potremmo semplicemente definire un valore di separazione per la caratteristica *sepal width* e porre la domanda binaria "sepal width ≥ 2.8 cm?".

Utilizzando l'algoritmo decisionale, iniziamo alla radice dell'albero e suddividiamo i dati in base alla caratteristica che produce il massimo guadagno informativo (*information gain – IG*), di cui parleremo più in dettaglio nel prossimo paragrafo. In un processo iterativo, possiamo ripetere questa procedura di suddivisione in ciascun nodo figlio, fino a giungere a foglie pure. Questo significa che i campioni in ciascun nodo appartengono tutti alla stessa classe. Nella pratica, ciò può produrre un albero molto profondo e dotato di molti nodi, il che può con facilità generare un problema di overfitting. Pertanto, quello che tipicamente dobbiamo fare è *potare* (*prune*) l'albero, impostando un limite alla sua profondità massima.

Massimizzare il guadagno informativo: la massima sostanza al minimo costo

Per poter suddividere i nodi sulla base delle caratteristiche maggiormente informative, dobbiamo definire una funzione obiettivo che vogliamo ottimizzare con l'algoritmo di apprendimento ad albero. Qui, la nostra funzione obiettivo tende a massimizzare il guadagno informativo a ogni suddivisione, che possiamo definire nel seguente modo:

$$IG\left(D_p, f\right) = I\left(D_p\right) - \sum_{j=1}^{m} \frac{N_j}{N_p} I\left(D_j\right)$$

Qui, *f* è la caratteristica sulla quale si basa la suddivisione, D_p e D_j sono il dataset del genitore e del j-esimo nodo figlio, *I* è la nostra misura di impurità, N_p è il numero totale di campioni nel nodo genitore e N_j è il numero di campioni nel nodo figlio j-esimo Come possiamo vedere, il guadagno informativo è semplicemente la differenza fra l'impurità del nodo genitore e la somma delle impurità dei nodi figli: minore è l'impurità dei nodi figli, maggiore è il guadagno informativo. Tuttavia, per semplicità e per ridurre lo spazio di ricerca combinatorio, la maggior parte delle librerie (fra cui scikit-learn) implementa alberi decisionali binari. Questo significa che da ogni nodo genitore dipendono due nodi figli, D_{left} e D_{right}:

$$IG\left(D_p, a\right) = I\left(D_p\right) - \frac{N_{left}}{N_p} I\left(D_{left}\right) - \frac{N_{right}}{N_p} I\left(D_{right}\right)$$

Ora, le tre misure di misure di impurità o criteri di suddivisione che vengono comunemente utilizzati negli alberi decisionali binari sono l'*impurità di Gini* (I_G), l'*entropia* (I_H) e l'*errore di classificazione* (I_E). Iniziamo definendo l'entropia per tutte le le classi *non vuote* $p(i|t) \neq 0$:

$$I_H\left(t\right) = -\sum_{i=1}^{c} p\left(i|t\right) \log_2 p\left(i|t\right)$$

Qui, $p(i|t)$ è la proporzione dei campioni che appartengono alla classe *c* per un determinato nodo *t*. L'entropia, pertanto, è 0 se tutti i campioni di un nodo appartengono alla stessa classe ed è massima se abbiamo una distribuzione uniforme nelle classi. Per esempio, nel caso dell'impostazione di una classe binaria, l'entropia è 0 se $p(i=1|t)=1$ o $p(i=0|t)=0$. Se le classi sono distribuite in modo uniforme con $p(i=1|t)=0.5$ e $p(i=0|t)=0.5$, l'entropia è pari a 1. Pertanto, possiamo dire che il criterio di entropia cerca di massimizzare l'informazione reciproca all'interno di un albero.

Intuitivamente, l'impurità di Gini può essere considerata come un criterio per minimizzare la probabilità di un'errata classificazione:

$$I_G\left(t\right) = \sum_{i=1}^{c} p\left(i|t\right)\left(1 - p\left(i|t\right)\right) = 1 - \sum_{i=1}^{c} p\left(i|t\right)^2$$

Come l'entropia, l'impurità di Gini è massima se le classi sono perfettamente mescolate, per esempio, in una classe binaria ($c=2$):

$$1 - \sum_{i=1}^{c} 0.5^2 = 0.5$$

Tuttavia, in pratica, sia l'impurità di Gini sia l'entropia forniscono in genere risultati molto simili e non vale la pena di dedicare loro molto tempo nella valutazione degli alberi utilizzando criteri di impurità differenti, ma piuttosto sperimentare con diversi criteri di potatura.

Un'altra misura dell'impurità è l'errore di classificazione:

$$I_E = 1 - \max\left\{p(i \mid t)\right\}$$

Questo è un criterio utile per la potatura, ma non è consigliabile per far crescere un albero decisionale, in quanto è meno sensibile ai cambiamenti nelle probabilità che i nodi appartengano a determinate classi. Possiamo illustrare il tutto osservando le due situazioni di suddivisione rappresentate nella Figura 3.17.

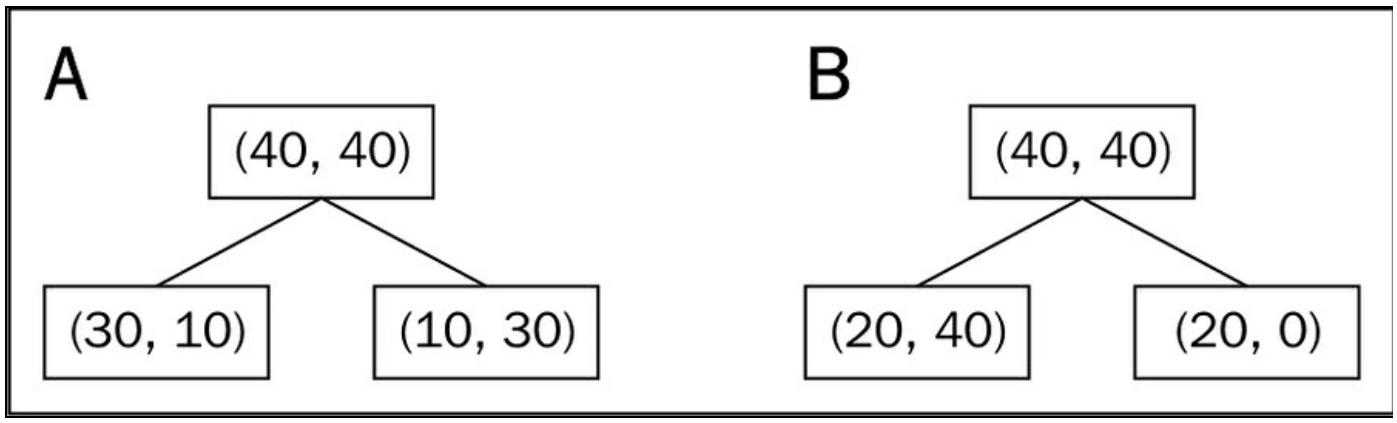


Figura 3.17

Partiamo con un dataset D_p al nodo genitore D_p costituito da 40 campioni della classe 1 e 40 campioni della classe 2 che dobbiamo suddividere nei due dataset D_{left} e D_{right}. Il guadagno informativo utilizzando l'errore di classificazione come criterio di suddivisione sarebbe lo stesso ($IG_E = 0.25$) in entrambe le situazioni, A e B:

$$I_E\left(D_p\right) = 1 - 0.5 = 0.5$$

$$A: I_E\left(D_{left}\right) = 1 - \frac{3}{4} = 0.25$$

$$A: I_E\left(D_{right}\right) = 1 - \frac{3}{4} = 0.25$$

$$A: IG_E = 0.5 - \frac{4}{8}0.25 - \frac{4}{8}0.25 = 0.25$$

$$B: I_E\left(D_{left}\right) = 1 - \frac{4}{6} = \frac{1}{3}$$

$$B: I_E\left(D_{right}\right)=1-1=0$$

$$B: IG_E=0.5-\frac{6}{8}\times\frac{1}{3}-0=0.25$$

Tuttavia, l'impurità di Gini preferirebbe la suddivisione della situazione $B\left(IG_G=0.1\overline{6}\right)$ rispetto alla situazione $A\left(IG_G=0.125\right)$, la quale è più *pura*:

$$I_G\left(D_p\right)=1-\left(0.5^2+0.5^2\right)=0.5$$

$$A: I_G\left(D_{left}\right)=1-\left(\left(\frac{3}{4}\right)^2+\left(\frac{1}{4}\right)^2\right)=\frac{3}{8}=0.375$$

$$A: I_G\left(D_{right}\right)=1-\left(\left(\frac{1}{4}\right)^2+\left(\frac{3}{4}\right)^2\right)=\frac{3}{8}=0.375$$

$$A: I_G=0.5-\frac{4}{8}0.375-\frac{4}{8}0.375=0.125$$

$$B: I_G\left(D_{left}\right)=1-\left(\left(\frac{2}{6}\right)^2+\left(\frac{4}{6}\right)^2\right)=\frac{4}{9}=0.\overline{4}$$

$$B: I_G\left(D_{right}\right)=1-\left(1^2+0^2\right)=0$$

$$B: IG_G=0.5-\frac{6}{8}0.\overline{4}-0=0.\overline{16}$$

Analogamente, il criterio di entropia preferirebbe la situazione $B\left(IG_H=0.31\right)$ rispetto alla situazione $A\left(IG_H=0.19\right)$:

$$I_H\left(D_p\right)=-\left(0.5\ \log_2\left(0.5\right)+0.5\ \log_2\left(0.5\right)\right)=1$$

$$A: I_H\left(D_{left}\right) = -\left(\frac{3}{4}\log_2\left(\frac{3}{4}\right) + \frac{1}{4}\log_2\left(\frac{1}{4}\right)\right) = 0.81$$

$$A: I_H\left(D_{right}\right) = -\left(\frac{1}{4}\log_2\left(\frac{1}{4}\right) + \frac{3}{4}\log_2\left(\frac{3}{4}\right)\right) = 0.81$$

$$A: IG_H = 1 - \frac{4}{8}0.81 - \frac{4}{8}0.81 = 0.19$$

$$B: I_H\left(D_{left}\right) = -\left(\frac{2}{6}\log_2\left(\frac{2}{6}\right) + \frac{4}{6}\log_2\left(\frac{4}{6}\right)\right) = 0.92$$

$$B: I_H\left(D_{right}\right) = 0$$

$$B: IG_H = 1 - \frac{6}{8}0.92 - 0 = 0.31$$

Per un confronto più intuitivo del funzionamento dei tre diversi criteri di impurità di cui abbiamo parlato, tracciamo gli indici di impurità per la gamma di probabilità [0, 1] per la classe 1. Notate che aggiungeremo anche una versione in scala dell'entropia (*entropia/2*) per osservare che l'impurità di Gini è una misura intermedia fra l'entropia e l'errore di classificazione. Il codice è il seguente:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                 ['Entropy', 'Entropy (scaled)',
...                 'Gini Impurity',
...                 'Misclassification Error'],
...                 ['-', '-', '--', '-.'],
...                 ['black', 'lightgray',
...                   'red', 'green', 'cyan']):
...     line = ax.plot(x, i, label=lab,
...                    linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...           ncol=3, fancybox=True, shadow=False)
```

```
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('Impurity Index')
>>> plt.show()
```

La Figura 3.18 è il risultato del codice precedente.

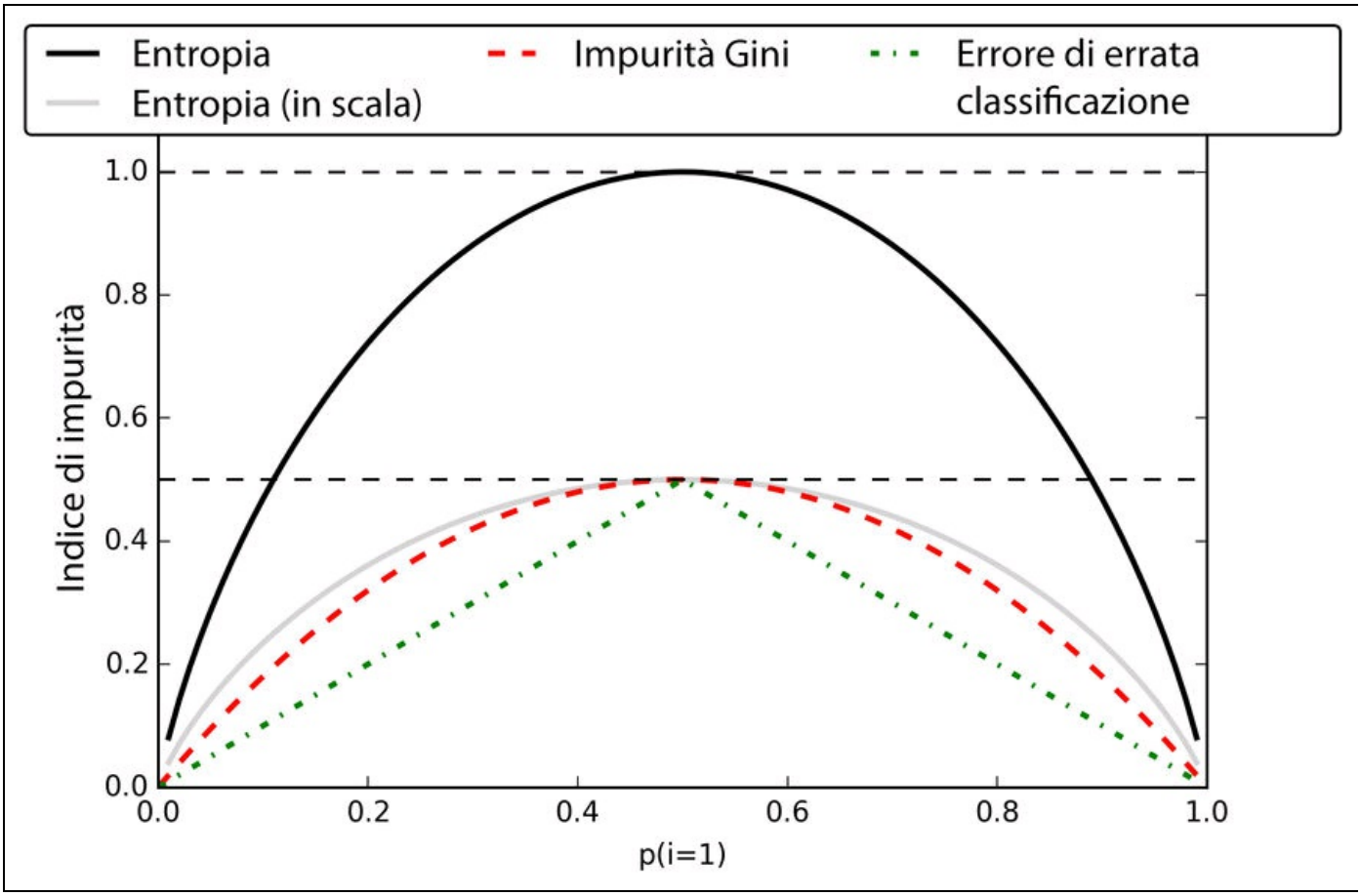


Figura 3.18

Costruire un albero decisionale

Gli alberi decisionali possono produrre complessi confini decisionali dividendo lo spazio delle caratteristiche in rettangoli. Tuttavia, dobbiamo fare attenzione, poiché più è profondo l'albero decisionale, più complesso diviene il confine decisionale, con un problema di overfitting. Utilizzando scikit-learn, addestreremo un albero decisionale con una massima profondità 3, utilizzando, come criterio di impurità, l'entropia. Sebbene l'adattamento in scala delle caratteristiche possa essere comodo per scopi di visualizzazione, notate che tale operazione non è un requisito degli algoritmi degli alberi decisionali. Ecco il codice utilizzato:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=3, random_state=0)
>>> tree.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
```

```
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined, y_combined,
...                       classifier=tree, test_idx=range(105,150))
>>> plt.xlabel('petal length [cm]')
>>> plt.ylabel('petal width [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Dopo l'esecuzione del codice precedente, otteniamo (Figura 3.19) i tipici confini ad assi paralleli di un albero decisionale.

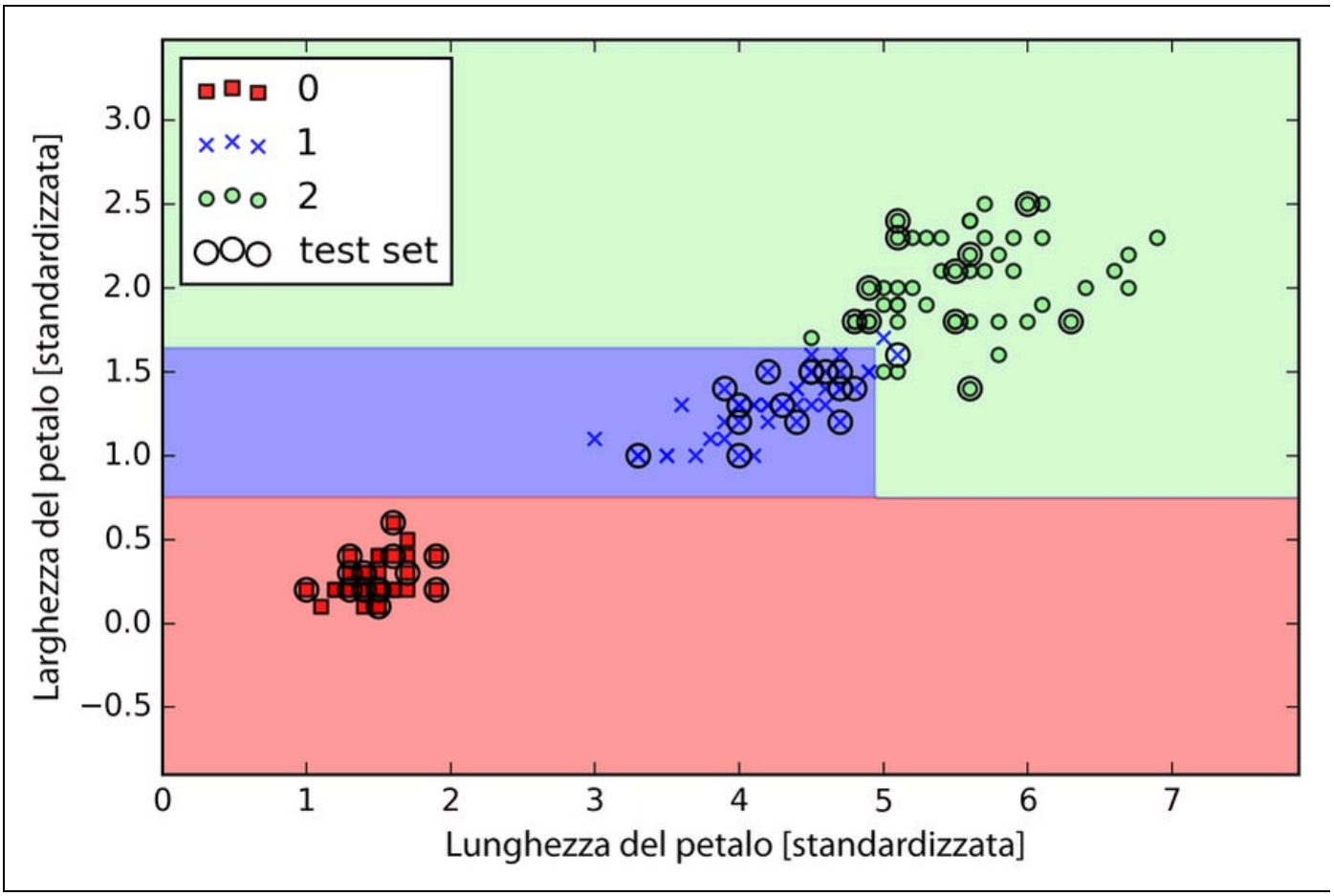


Figura 3.19

Un'ottima funzionalità di scikit-learn è la possibilità di esportare l'albero decisionale come un file `.dot` dopo l'addestramento, in modo che possiamo visualizzarlo utilizzando il programma GraphViz. Questo programma è liberamente disponibile all'indirizzo http://www.graphviz.org ed è disponibile per Linux, Windows e Mac OS X.

Innanzitutto creiamo il file `.dot` tramite scikit-learn, utilizzando la funzione `export_graphviz` del modulo `tree`, nel seguente modo:

```
>>> from sklearn.tree import export_graphviz
>>> export_graphviz(tree,
...                 out_file='tree.dot',
...                 feature_names=['petal length', 'petal width'])
```

Dopo aver installato GraphViz sul computer, possiamo convertire il file tree.dot in un file PNG lanciando il seguente comando dalla riga di comando nella cartella in cui abbiamo salvato il file tree.dot:

```
> dot -Tpng tree.dot -o tree.png
```

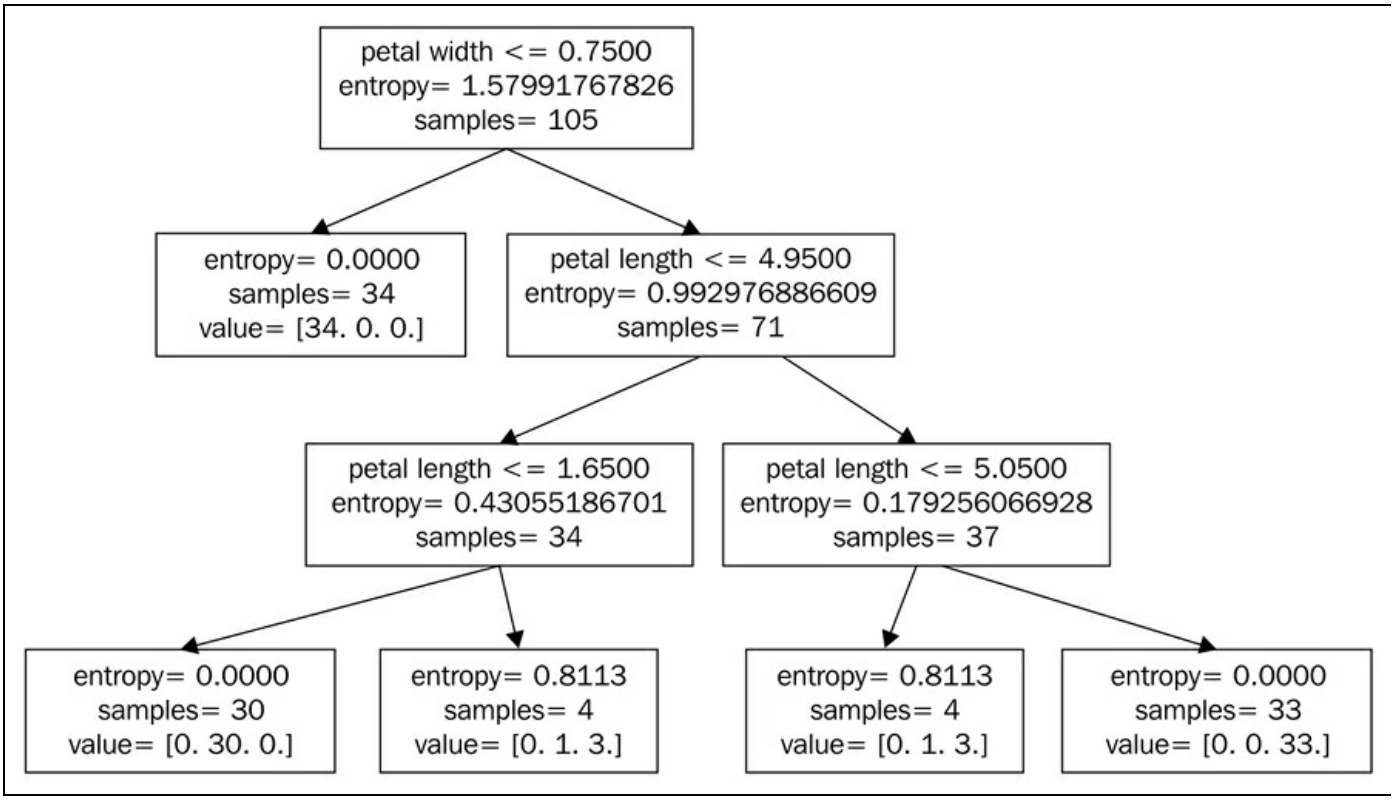


Figura 3.20

Osservando la Figura 3.20, che rappresenta l'albero decisionale che abbiamo creato con GraphViz, possiamo ora tracciare con comodità le suddivisioni determinate dall'albero decisionale sul dataset di addestramento. Siamo partiti con 105 campioni alla radice e abbiamo suddiviso due nodi figli, con 34 e 71 campioni ciascuno, utilizzando per la separazione la larghezza dei petali (*petal width* ≤ 0.75 cm). Dopo la prima suddivisione, possiamo vedere che il nodo a sinistra è già puro e contiene solo campioni della classe Iris-Setosa (entropia uguale a 0). Le ulteriori suddivisioni che avvengono sul lato destro servono a separare i campioni appartenenti alle classi Iris-Versicolor e Iris-Virginica.

Combinare sistemi di apprendimento deboli e forti tramite foreste casuali

Le *foreste casuali* hanno acquisito grande popolarità nelle applicazioni di machine learning durante l'ultimo decennio grazie alle loro buone prestazioni di

classificazione, alla loro scalabilità e alla loro facilità d'uso. Intuitivamente, una foresta casuale può essere considerata come un *insieme* di alberi decisionali. L'idea che spinge a utilizzare un insieme di sistemi di apprendimento consiste nel combinare più *sistemi di apprendimento deboli* per costruire un modello più robusto, un *sistema di apprendimento forte* che offra un migliore errore di generalizzazione e sia meno suscettibile a problemi di overfitting. L'algoritmo a foresta casuale può essere riepilogato in quattro semplici passi.

1. Trarre un campione casuale iniziale (*bootstrap*) di dimensioni *n* (scegliete casualmente *n* campioni dal set di addestramento con reinserimento).
2. Far crescere un albero decisionale dal campione di bootstrap. Per ogni nodo:
    1. selezionare casualmente *d* caratteristiche senza reinserimento;
    2. suddividere il nodo utilizzando la caratteristica che fornisce la migliore suddivisione sulla base della funzione obiettivo, per esempio massimizzando il guadagno informativo.
3. Ripetere per *k* volte i passi 1 e 2.
4. Aggregare le previsioni di ciascun albero per assegnare l'etichetta della classe sulla base di un voto a maggioranza. Il voto a maggioranza verrà trattato più in dettaglio nel Capitolo 7, *Combinare più modelli: l'apprendimento d'insieme*.

Vi è una leggera modifica al Passo 2 quando si devono addestrare singoli alberi decisionali: invece di valutare tutte le funzioni sulle caratteristiche per determinare la migliore suddivisione in ciascun nodo, considereremo solo un sottoinsieme casuale di questi.

Sebbene le foreste casuali non offrano lo stesso livello di interoperabilità degli alberi decisionali, un loro grande vantaggio è il fatto che non dobbiamo preoccuparci troppo della scelta di un buon valore per gli iperparametri. In generale non dobbiamo potare la foresta casuale, poiché nel suo insieme il modello è piuttosto resistente al rumore rispetto ai singoli alberi decisionali. L'unico parametro di cui dobbiamo davvero preoccuparci, in pratica, è il numero di alberi *k* (Passo 3) che abbiamo scelto per la foresta casuale. Tipicamente, maggiore è il numero di alberi, migliori saranno le prestazioni del classificatore a foresta casuale, con lo svantaggio di un incremento del costo computazionale.

Sebbene sia meno comuni nell'uso pratico, vi sono altri iperparametri del classificatore a foresta casuale che possono essere utilizzati, tramite tecniche di cui parleremo nel Capitolo 5, *Compressione dei dati tramite la riduzione della dimensionalità*: si tratta delle dimensioni *n* del campione di bootstrap (Passo 1) e

del numero di caratteristiche *d* che vengono scelte casualmente per ciascuna suddivisione (Passo 2.1). Con le dimensioni *n* del campione di bootstrap, controlliamo il compromesso bias-varianza della foresta casuale. Scegliendo un valore più grande di *n*, riduciamo la casualità e pertanto la foresta è più probabile che vada in overfit. Al contrario, possiamo ridurre il livello di overfitting scegliendo valori più piccoli per *n*, riducendo però le prestazioni del modello, in termini di velocità. Nella maggior parte delle implementazioni, compresa l'implementazione RandomForestClassifier di scikit-learn, le dimensioni del campione di bootstrap vengono scelte in modo che siano uguali al numero di campioni del set di addestramento originario, il che rappresenta un buon compromesso fra varianza e bias. Per il numero di caratteristiche *d* a ciascuna suddivisione, vogliamo scegliere un valore che sia più compatto del numero totale di caratteristiche presenti nel set di addestramento. Un valore predefinito ragionevole che viene utilizzato in scikit-learn e altre implementazioni è $d = \sqrt{m}$, dove *m* è il numero di caratteristiche presenti nel set di addestramento.

Fortunatamente non dobbiamo costruire il classificatore a foresta casuale manualmente utilizzando singoli alberi decisionali; scikit-learn offre già un'implementazione pronta all'uso:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(criterion='entropy',
...                                 n_estimators=10,
...                                 random_state=1,
...                                 n_jobs=2)
>>> forest.fit(X_train, y_train)
>>> plot_decision_regions(X_combined, y_combined,
...                       classifier=forest, test_idx=range(105,150))
>>> plt.xlabel('petal length')
>>> plt.ylabel('petal width')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Dopo aver eseguito il codice precedente, dovremmo vedere le regioni decisionali formate dall'insieme di alberi presenti nella foresta casuale, come illustrato nella Figura 3.21.

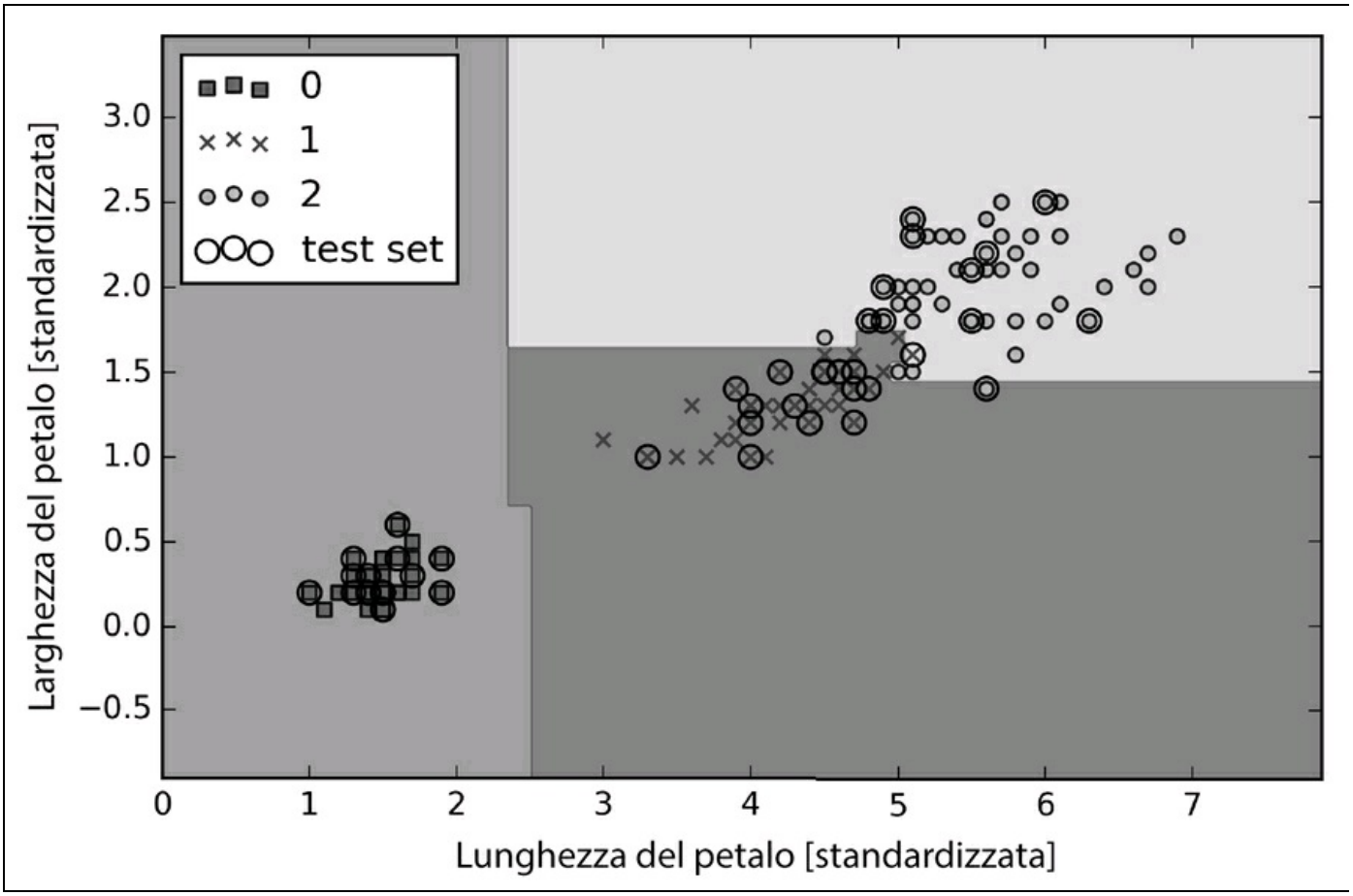


Figura 3.21

Utilizzando il codice precedente, abbiamo addestrato una foresta casuale di dieci alberi decisionali tramite il parametro `n_estimators` e, come misurazione dell'impurità per suddividere i nodi, abbiamo utilizzato il criterio di entropia. Anche se facciamo crescere una piccolissima foresta casuale da un piccolissimo dataset di addestramento, abbiamo utilizzato il parametro `n_jobs` per scopi dimostrativi, per consentirci di parallelizzare l'addestramento del modello utilizzando i core disponibili nel computer (in questo caso due).

I k vicini più prossimi: un algoritmo di apprendimento pigro

L'ultimo algoritmo di apprendimento con supervisione di cui vogliamo parlare in questo capitolo si chiama *KNN* (*k-nearest neighbor classifier*, in pratica "classificatore a k vicini più prossimi") ed è particolarmente interessante poiché è fondamentalmente differente dagli algoritmi di apprendimento di cui abbiamo parlato finora.

KNN è un tipico esempio di sistema di apprendimento pigro. Si dice *pigro* (*lazy*) non per la sua apparente semplicità, ma perché non apprende una funzione di discriminazione dai dati di addestramento, ma piuttosto memorizza il dataset di addestramento.

Approfondimento

Gli algoritmi di machine learning possono essere suddivisi in modelli *parametrici* e *non parametrici*. Utilizzando modelli parametrici, possiamo stimare i parametri dal dataset di addestramento per ottenere una funzione in grado di classificare nuovi dati senza richiedere più il dataset di addestramento originale. Tipici esempi di modelli parametrici sono il perceptron, la regressione logistica e la SVM lineare. Al contrario, i modelli non parametrici non possono essere caratterizzati da un numero fisso di parametri: il numero dei parametri cresce sulla base dei dati di addestramento. Due esempi di modelli non parametrici che abbiamo visto finora sono il classificatore ad albero decisionale/a foresta casuale e la SVM kernel.

KNN appartiene a una sottocategoria dei modelli non parametrici descritta come *apprendimento basato su istanze*. I modelli con apprendimento basato su istanze sono caratterizzati dalla memorizzazione del dataset di addestramento e l'apprendimento pigro è un caso speciale di apprendimento basato su istanze associato a costo zero durante il processo di apprendimento.

L'algoritmo KNN è molto semplice e può essere riassunto nei passi seguenti.

1. Scegliere il numero k e una metrica della distanza.
2. Trovare i k elementi più vicini del campione che vogliamo classificare.
3. Assegnare l'etichetta della classe sulla base di un voto a maggioranza.

La Figura 3.22 illustra come un nuovo punto nei dati (?) venga assegnato alla classe dei triangoli sulla base del voto di maggioranza fra i suoi cinque vicini più prossimi.

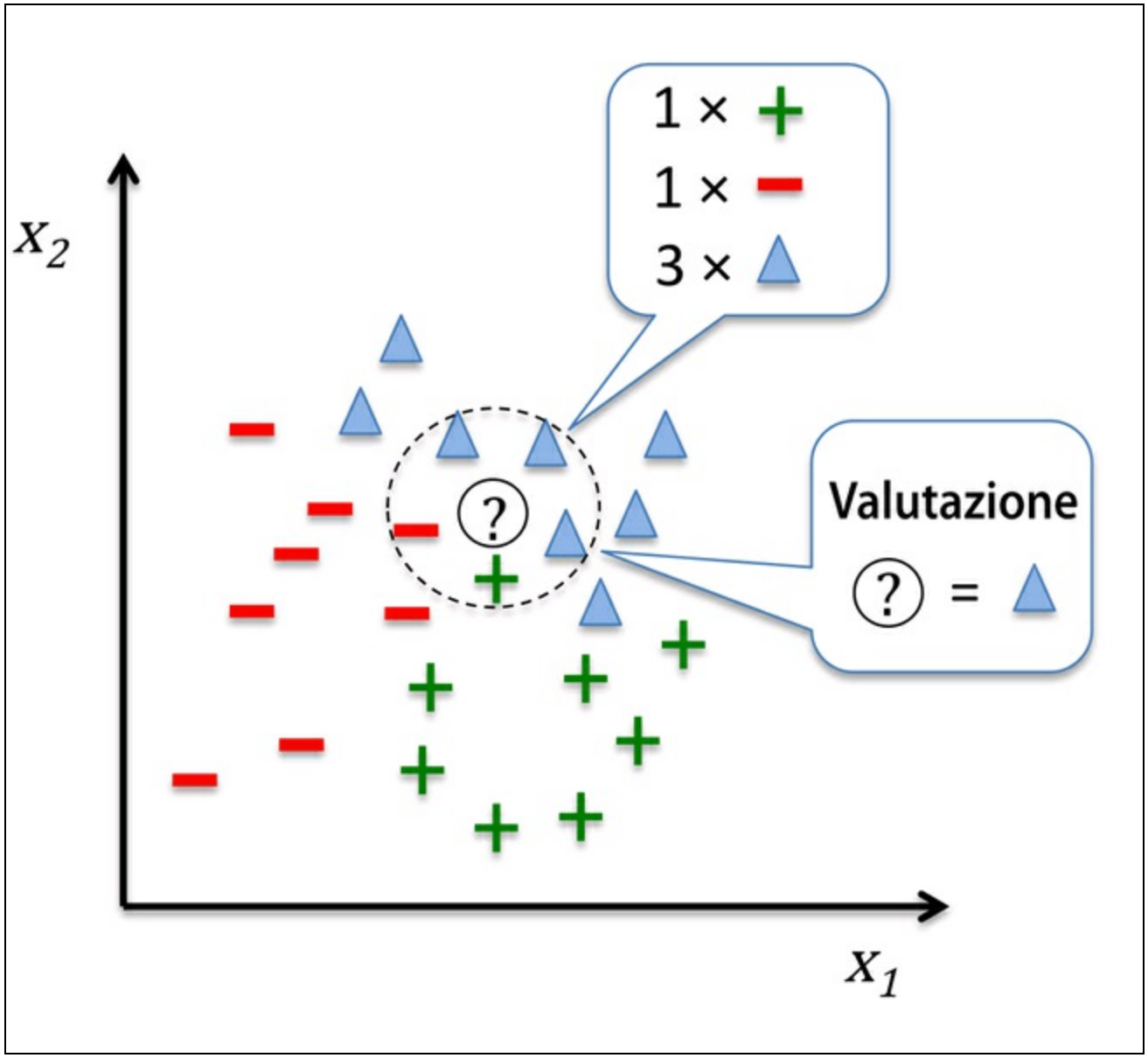


Figura 3.22

Sulla base della metrica di distanza scelta, l'algoritmo KNN trova i *k* campioni del dataset di addestramento che sono più vicini (più simili) al punto che intendiamo classificare. L'etichetta della classe del nuovo punto viene poi determinata sulla base di un voto a maggioranza fra i suoi *k* vicini più prossimi.

Il vantaggio principale di questo approccio basato sulla memorizzazione è il fatto che il classificatore si adatta immediatamente mentre raccogliamo nuovi dati di addestramento. Tuttavia, lo svantaggio è il fatto che la complessità computazionale per la classificazione di nuovi campioni cresce in modo lineare con il numero di campioni presenti nel dataset di addestramento (nella situazione peggiore) a meno che il dataset abbia pochissime dimensioni (caratteristiche) e l'algoritmo sia stato implementato utilizzando strutture dati efficienti come i KD-trees (J. H. Friedman, J. L. Bentley e R. A. Finkel. *An algorithm for finding best matches in logarithmic*

expected time. “ACM Transactions on Mathematical Software (TOMS)”, 3(3):209–226, 1977). Inoltre, non possiamo eliminare i campioni di addestramento, in quanto non esiste un passo di *addestramento*. Pertanto, lo spazio di memorizzazione può diventare un problema se ci troviamo a lavorare con dataset di grandi dimensioni.

Il seguente codice implementa un modello KNN in scikit-learn utilizzando una metrica di distanza euclidea:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                            metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                       classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.show()
```

Specificando cinque vicini nel modello KNN per questo dataset, otteniamo un confine decisionale relativamente morbido, come possiamo vedere nella Figura 3.23.

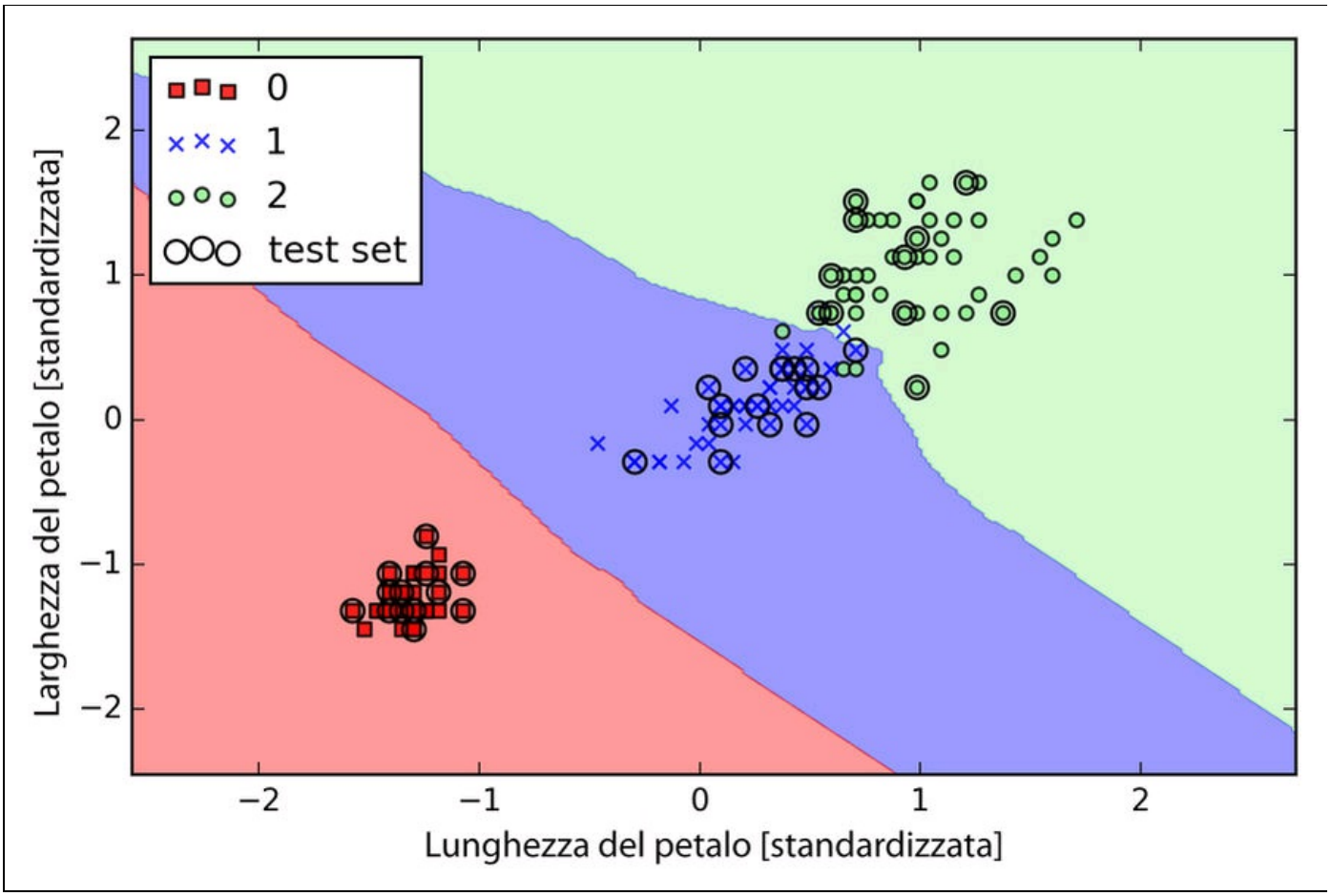


Figura 3.23

NOTA

Nel caso di un nodo, l’implementazione scikit-learn dell’algoritmo KNN preferirà i vicini che hanno minore distanza dal campione. Se i vicini hanno una distanza simile, l’algoritmo sceglierà l’etichetta della classe che compare per prima nel dataset di addestramento.

La scelta *corretta* di *k* è fondamentale per trovare un buon equilibrio fra i problemi di over- e under-fitting. Inoltre dobbiamo assicurarci di aver scelto una metrica della distanza che sia appropriata per la caratteristica del dataset. Spesso, per i campioni del mondo reale, per esempio i fiori del dataset Iris, i quali hanno caratteristiche misurate in centimetri, viene utilizzata una semplice misurazione della distanza euclidea. Tuttavia, se utilizziamo una misurazione della distanza euclidea, è anche importante standardizzare i dati in modo che ogni caratteristica contribuisca in modo uniforme alla distanza. La distanza `'minkowski'` che abbiamo utilizzato nel codice precedente è semplicemente una generalizzazione delle distanze euclidea e Manhattan che può essere scritta nel seguente modo:

$$d\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(i)}\right) = \sqrt[p]{\sum_k \left|x_k^{(i)} - x_k^{(j)}\right|^p}$$

Questa diviene una distanza euclidea se impostiamo il parametro `p=2` o la distanza mancante a `p=1`. Molte altre valutazioni di distanza sono disponibili in scikit-learn e possono essere fornite al parametro `metric`. Sono elencate in http://scikitlearn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html.

NOTA

è importante menzionare che l'algoritmo KNN è molto suscettibile al problema di overfitting, a causa della maledizione della dimensionalità. Si tratta di un fenomeno in cui lo spazio delle caratteristiche diviene sempre più sparso a causa della crescita di dimensioni di un dataset di addestramento di dimensioni fisse. Intuitivamente, possiamo considerare che anche i vicini più prossimi siano troppo lontani, in uno spazio di grandi dimensioni, per poter offrire una buona stima. Abbiamo parlato del concetto di regolarizzazione nel paragrafo dedicato alla regressione logistica come a un modo per evitare il problema dell'overfitting. Tuttavia, nei modelli in cui la regolarizzazione non è applicabile, come negli alberi decisionali e nei KNN, possiamo utilizzare alcune tecniche di selezione delle caratteristiche e di riduzione della dimensionalità per cercare di sfuggire alla maledizione della dimensionalità. L'argomento verrà trattato più in dettaglio nel prossimo capitolo.

Riepilogo

In questo capitolo abbiamo trattato vari algoritmi utilizzati per affrontare problemi lineari e non lineari. Abbiamo visto che gli alberi decisionali sono particolarmente interessanti se ci interessa l'interpretabilità. La regressione logistica non è solo un modello utile per l'apprendimento online o l'apprendimento a gradiente stocastico, ma ci consente anche di prevedere la probabilità di un determinato evento. Sebbene le macchine a vettori di supporto siano modelli lineari potenti, che possono essere estesi a problemi non lineari, o con la tecnica kernel, impiegano troppi parametri, che devono essere ottimizzati per ottenere buone previsioni. Al contrario, i metodi d'insieme, come le foreste casuali, non richiedono particolare ottimizzazione dei parametri e non subiscono il problema dell'overfit con la stessa facilità degli alberi decisionali, il che li rende un modello interessante per il dominio dei problemi pratici. Il classificatore dei k vicini più prossimi (KNN) offre un approccio alternativo alla classificazione tramite l'apprendimento pigro, che ci consente di effettuare previsioni senza alcun addestramento del modello, ma con un passo di previsione più costoso dal punto di vista computazionale.

Tuttavia, ancora più importante rispetto alla scelta di un algoritmo di apprendimento appropriato è quello relativo ai dati disponibili nel dataset di apprendimento. Nessun algoritmo sarà in grado di eseguire previsioni di qualità senza caratteristiche informative e discriminatorie.

Nel prossimo capitolo tratteremo argomenti importanti che riguardano la pre-elaborazione dei dati, la scelta delle caratteristiche e la riduzione della dimensionalità, tutte tecniche utili per costruire modelli più potenti di apprendimento. Successivamente, nel Capitolo 6, *Valutazione dei modelli e ottimizzazione degli iperparametri*, vedremo come possiamo valutare e confrontare le prestazioni dei nostri modelli e applicare tecniche utili per ottimizzare i diversi algoritmi.

Costruire buoni set di addestramento: la pre-elaborazione

La qualità dei dati e la quantità di informazioni utili che essi contengono sono fattori chiave per determinare la qualità di apprendimento di un algoritmo di machine learning. Pertanto, è assolutamente fondamentale assicurarsi di esaminare e pre-elaborare un dataset prima di fornirlo a un algoritmo di apprendimento. In questo capitolo, tratteremo le principali tecniche di pre-elaborazione dei dati, che ci aiuteranno a costruire modelli di apprendimento migliori.

Gli argomenti che tratteremo nel corso del capitolo sono i seguenti.

- Rimozione e imputazione dei valori mancanti da un dataset.
- Miglioramento della collocazione in categorie dei dati per gli algoritmi di machine learning.
- Selezione delle caratteristiche rilevanti per la costruzione di un modello.

Il problema dei dati mancanti

Nelle applicazioni del mondo reale, è normale che nei campioni manchino uno o più valori, per vari motivi. La causa potrebbe essere un errore nel processo di raccolta dei dati oppure determinate misurazioni non sono applicabili, determinati campi sono stati trascurati in una rilevazione e così via. Tipicamente individuiamo i *valori mancanti* in termini di spazi vuoti nella tabella dei dati oppure come stringhe segnaposto, per esempio valori NaN (Not A Number).

Sfortunatamente, la maggior parte degli strumenti computazionali non è in grado di gestire tali valori mancanti o produrrebbe risultati imprevedibili se decidessimo di ignorarli. Pertanto, è fondamentale occuparsi dei valori mancanti prima di procedere oltre con l'analisi. Ma prima di considerare le varie tecniche impiegabili per gestire il problema dei valori mancanti, creiamo un semplice insieme di dati d'esempio su un file *CSV* (*comma-separated values*) che consenta di valutare meglio il problema:

```
>>> import pandas as pd
>>> from io import StringIO
>>> csv_data = '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''
>>> # If you are using Python 2.7, you need
>>> # to convert the string to unicode:
>>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>>> df
   A   B   C   D
0  1   2   3   4
1  5   6 NaN   8
2  10  11  12 NaN
```

Utilizzando il codice precedente, leggiamo i dati all'interno di un pandas DataFrame tramite la funzione read_csv e notiamo che le due celle mancanti sono state sostituite da valori NaN. La funzione StringIO dell'esempio di codice precedente è stata utilizzata solo per scopi di illustrazione. Consente di leggere la stringa assegnata a csv_data in un pandas DataFrame come se fosse un vero file CSV sul disco rigido.

Per un DataFrame di dimensioni non banali, può essere noioso ricercare manualmente i valori mancanti. In questo caso, utilizzeremo il metodo isnull per restituire un DataFrame con valori booleani che indicano se una cella contiene un valore numerico (False) o se i dati sono mancanti (True). Utilizzando il metodo sum, possiamo pertanto restituire il numero di valori mancanti per colonna nel seguente modo:

```
>>> df.isnull().sum()
A   0
B   0
```

```
C   1
D   1
dtype: int64
```

In questo modo possiamo contare il numero di valori mancanti per colonna; nei seguenti paragrafi esamineremo le varie strategie per gestire questi dati mancanti.

Approfondimento

Anche se scikit-learn è stato sviluppato per lavorare su array NumPy, talvolta può essere più comodo pre-elaborare i dati utilizzando pandas `DataFrame`. Possiamo sempre accedere all'array NumPy sottostante di `DataFrame` tramite l'attributo `values` prima di inviare i dati a un valutatore scikit-learn:

```
>>> df.values
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  nan,   8.],
       [ 10.,  11.,  12.,  nan]])
```

Eliminazione dei campioni e delle caratteristiche con valori mancanti

Uno dei modi più semplici per gestire il problema dei valori mancanti consiste nell'eliminare completamente dal dataset le caratteristiche (colonne) o i campioni (righe) corrispondenti; le righe con valori mancanti possono essere eliminate con facilità tramite il metodo `dropna`:

```
>>> df.dropna()
   A  B  C  D
0  1  2  3  4
```

Analogamente, possiamo eliminare le colonne che hanno almeno un valore `NaN` in una riga, impostando l'argomento `axis` a 1:

```
>>> df.dropna(axis=1)
   A   B
0  1   2
1  5   6
2  10  11
```

Il metodo `dropna` supporta vari parametri aggiuntivi che possono tornare utili:

```
# only drop rows where all columns are NaN
>>> df.dropna(how='all')

# drop rows that have not at least 4 non-NaN values
>>> df.dropna(thresh=4)

# only drop rows where NaN appear in specific columns (here: 'C')
>>> df.dropna(subset=['C'])
```

Sebbene la rimozione dei dati mancanti possa sembrare un approccio comodo, presenta anche una serie di svantaggi. Per esempio, potremmo finire per rimuovere fin troppi campioni, il che potrebbe pregiudicare del tutto l'affidabilità. Oppure, se rimuoviamo troppe colonne di caratteristiche, corriamo il rischio di perdere informazioni preziose, delle quali il nostro classificatore ha bisogno per poter

discriminare le classi. Nel prossimo paragrafo esamineremo, pertanto, una delle alternative più comunemente utilizzate per gestire il problema dei valori mancanti: le tecniche di interpolazione.

Imputazione dei valori mancanti

Spesso, la rimozione dei campioni o l'eliminazione di intere colonne di caratteristiche non è una via percorribile, perché potremmo perdere troppi dati preziosi. In questo caso, possiamo utilizzare varie tecniche di interpolazione per stimare i valori mancanti sulla base degli altri campioni del dataset. Una delle tecniche di interpolazione più comuni è l'*imputazione media*, mediante la quale sostituiamo semplicemente il valore mancante con il valore medio dell'intera colonna di caratteri della caratteristica. Un modo comodo per ottenere ciò consiste nell'utilizzare la classe `Imputer` di scikit-learn, come vediamo nel codice seguente:

```
>>> from sklearn.preprocessing import Imputer
>>> imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imr = imr.fit(df)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
array([[ 1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.5,  8.],
       [ 10.,  11.,  12.,  6.]])
```

Qui abbiamo sostituito ogni valore `NaN` con la relativa media, che viene calcolata separatamente per ogni colonna di caratteristiche. Se al posto di `axis=0` usiamo `axis=1`, calcoliamo la media della riga. Fra le opzioni per il parametro `strategy` vi sono `median` o `most_frequent`; quest'ultima sostituisce i valori mancanti con i valori più frequenti. Questo può essere utile per l'imputazione di valori della caratteristica.

Funzionamento dell'API di stima di scikit-learn

Nel paragrafo precedente, abbiamo utilizzato la classe `Imputer` di scikit-learn per imputare i valori mancanti nel dataset. La classe `Imputer` appartiene alle cosiddette classi *transformer* di scikit-learn, utilizzate per la trasformazione dei dati. I due metodi essenziali di questi estimatori sono `fit` e `transform`. Il metodo `fit` viene utilizzato per apprendere i parametri dai dati di addestramento e il metodo `transform` utilizza questi parametri per trasformare i dati. Ogni array di dati che deve essere trasformato deve avere lo stesso numero di caratteristiche dell'array di dati che è stato utilizzato per preparare il modello. La Figura 4.1 illustra come un trasformatore predisposto sui dati di addestramento possa essere utilizzato per trasformare un dataset di addestramento e anche un nuovo dataset di test.

I classificatori che abbiamo utilizzato nel Capitolo 3, *I classificatori di machine learning di scikit-learn*, appartengono ai cosiddetti stimatori di scikit-learn con un'API che, concettualmente, è molto simile alla classe transformer. Gli stimatori hanno un metodo `predict`, ma possono anche avere un metodo `transform`, come vedremo più avanti. Come ricorderete, abbiamo utilizzato anche il metodo `fit` per imparare i parametri di un modello quando abbiamo addestrato tali estimatori per la classificazione. Tuttavia, nei compiti di apprendimento con supervisione, in più forniamo le etichette delle classi per la configurazione del modello, che possono poi essere utilizzati per eseguire previsioni sui nuovi campioni di dati tramite il metodo `predict`, come illustrato nella Figura 4.2.

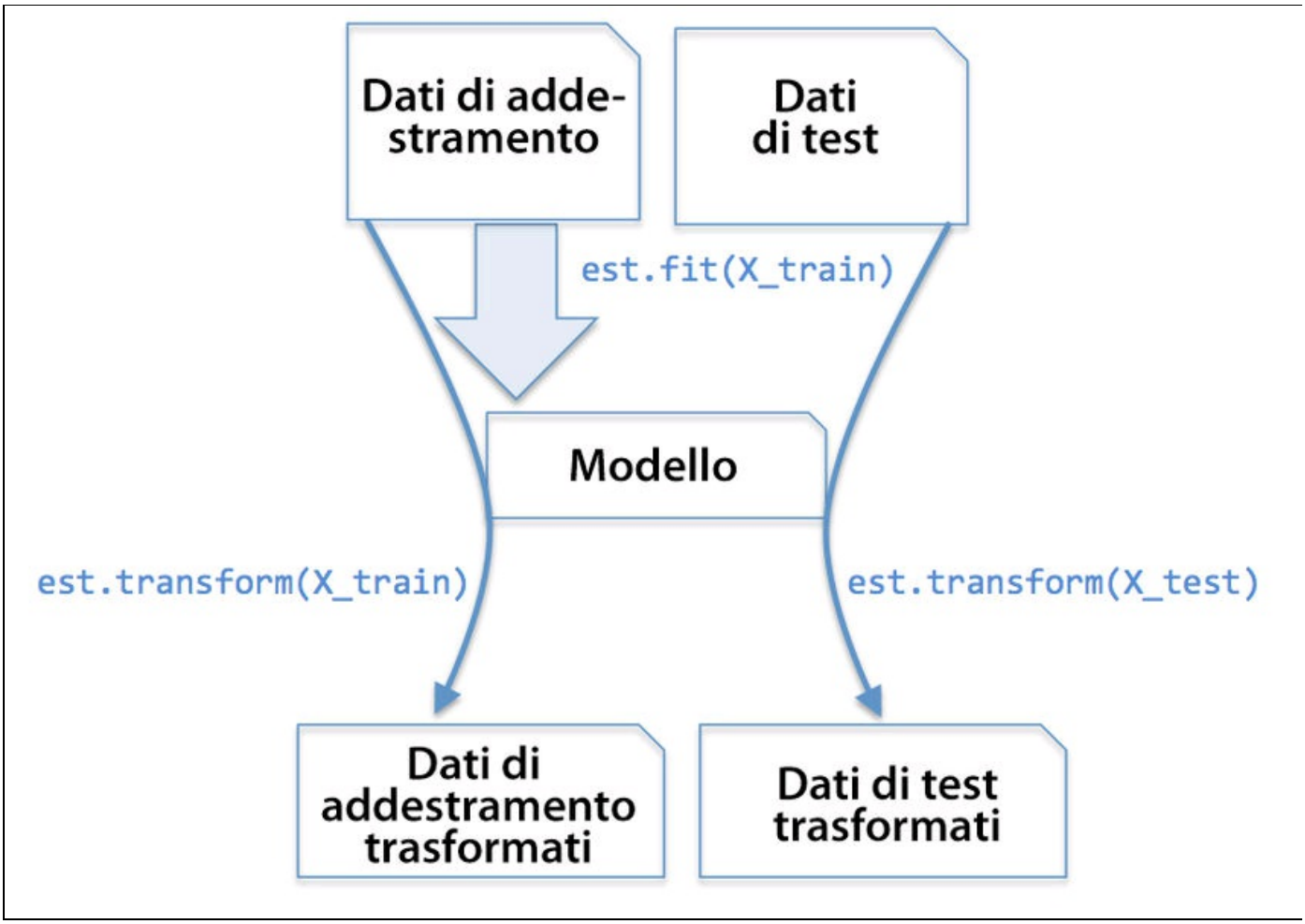


Figura 4.1

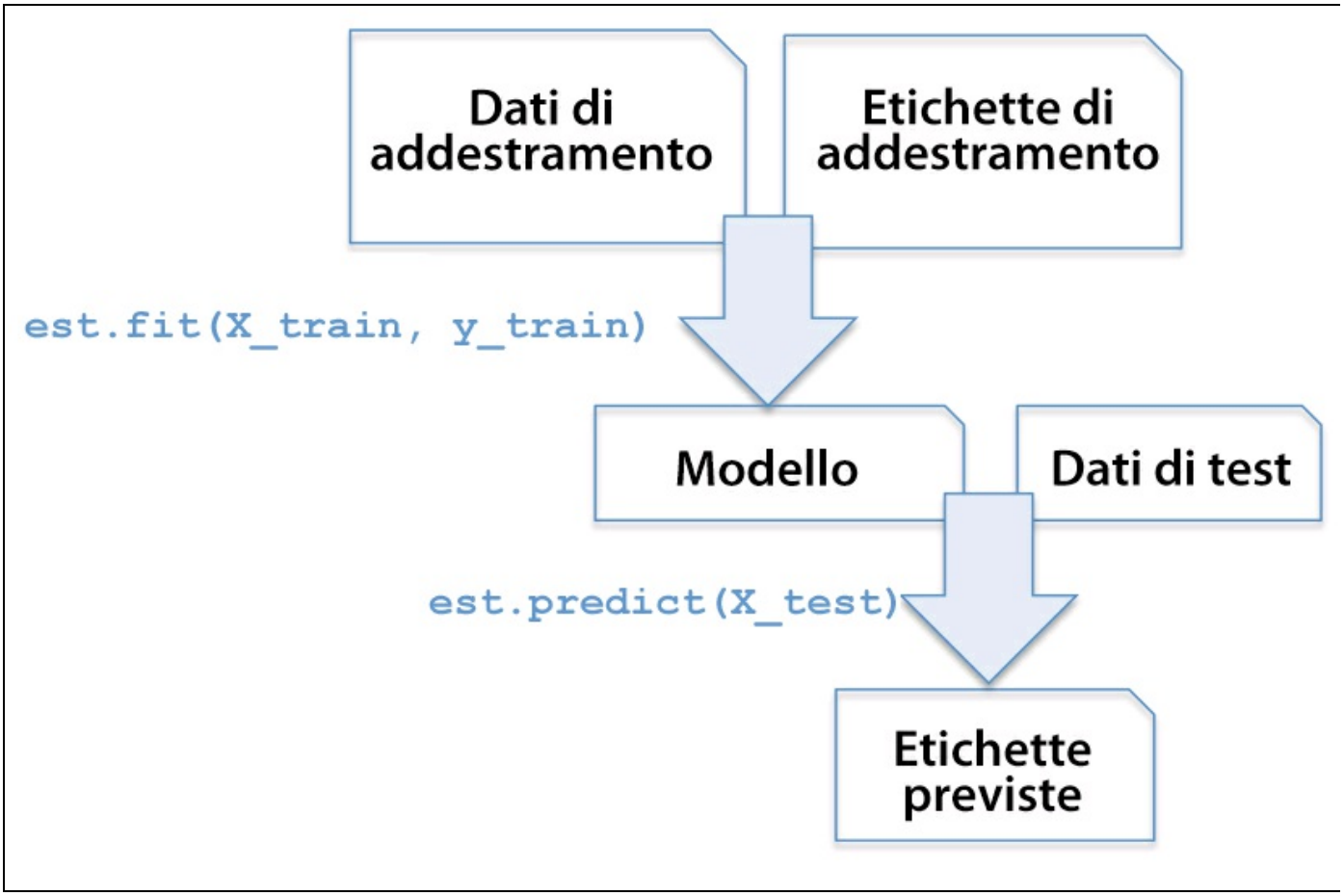

Dati di addestramento
Etichette di addestramento
est.fit(X_train, y_train)
Modello
Dati di test
est.predict(X_test)
Etichette previste

Figura 4.2

Gestione di dati categorici

Finora abbiamo lavorato solo su valori numerici. Tuttavia è abbastanza comune che i dataset del mondo reale contengano una o più colonne di caratteristiche categoriche. Quando parliamo di dati categorici, dobbiamo ulteriormente distinguere fra *caratteristiche nominali e ordinali*. Le caratteristiche ordinali sono valori categorici che possono essere ordinati. Per esempio, *T-shirt size* sarebbe una caratteristica ordinale, poiché possiamo definire un ordine *XL* > *L* > *M*. Al contrario, le caratteristiche nominali non prevedono alcun ordine e, con riferimento all'esempio precedente, potremmo considerare *T-shirt color* come una caratteristica nominale, in quanto, in genere, non ha senso dire che, per esempio, il *rosso* è maggiore del *blu*.

Prima di esplorare le varie tecniche per gestire tali dati categorici, creiamo un nuovo insieme di dati per illustrare il problema:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...           ['green', 'M', 10.1, 'class1'],
...           ['red', 'L', 13.5, 'class2'],
...           ['blue', 'XL', 15.3, 'class1']])
>>> df.columns = ['color', 'size', 'price', 'classlabel']
>>> df
   color size  price classlabel
0  green    M   10.1     class1
1    red    L   13.5     class2
2   blue   XL   15.3     class1
```

Come possiamo vedere nell'output precedente, il DataFrame appena creato contiene una caratteristica nominale (color), una caratteristica ordinale (size) e una caratteristica numerica (price), disposte per colonne. Le etichette delle classi (supponendo di aver creato un dataset per un compito di apprendimento con supervisione) sono conservate nell'ultima colonna. Gli algoritmi di classificazione ad apprendimento di cui parleremo in questo libro non utilizzano informazioni ordinali nelle etichette delle classi.

Mappaggio di caratteristiche ordinali

Per assicurarsi che gli algoritmi di apprendimento interpretino correttamente le caratteristiche ordinali, occorre convertire i valori stringa categorici in valori interi. Sfortunatamente, non esiste una funzione comoda in grado di derivare automaticamente l'ordine corretto delle etichette della caratteristica size. Pertanto,

dobbiamo definire il mappaggio in modo manuale. Nel seguente semplice esempio, supponiamo di conoscere la differenza fra le caratteristiche, per esempio

$$XL = L + 1 = M + 2$$

```
>>> size_mapping = {
...                 'XL': 3,
...                 'L': 2,
...                 'M': 1}
>>> df['size'] = df['size'].map(size_mapping)
>>> df
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```

Se, in un secondo tempo, vogliamo ritrasformare i valori interi nella loro rappresentazione a stringa originale, possiamo semplicemente definire un dizionario di mappaggio inverso `inv_size_mapping = {v: k for k, v in size_mapping.items()}` che può essere utilizzato tramite il metodo `map` di pandas sulla colonna della caratteristica trasformata, un po' come nel caso del dizionario `size_mapping` che abbiamo utilizzato in precedenza.

Codifica delle etichette delle classi

Molte librerie di machine learning richiedono che le etichette delle classi siano codificate come valori interi. Sebbene la maggior parte degli estimatori per la classificazione di scikit-learn converta internamente le etichette delle classi in valori interi, è considerata buona norma fornire le etichette delle classi come array di interi per evitare incomprensioni tecniche. Per codificare le etichette delle classi, possiamo utilizzare un approccio simile a quello del mappaggio delle caratteristiche ordinali, di cui abbiamo parlato in precedenza. Dobbiamo ricordare che le etichette delle classi *non sono* ordinali e non importa quale numero intero assegnamo a una determinata etichetta (stringa). Pertanto, possiamo semplicemente enumerare le etichette delle classi a partire da 0:

```
>>> import numpy as np
>>> class_mapping = {label:idx for idx,label in
...                 enumerate(np.unique(df['classlabel']))}
>>> class_mapping
{'class1': 0, 'class2': 1}
```

Poi possiamo utilizzare il dizionario di mappaggio per trasformare le etichette delle classi in valori interi:

```
>>> df['classlabel'] = df['classlabel'].map(class_mapping)
>>> df
   color  size  price  classlabel
0  green     1   10.1           0
1    red     2   13.5           1
2   blue     3   15.3           0
```

Possiamo invertire le coppie chiave-valore nel dizionario di mappaggio nel seguente modo per mappare le etichette (convertite) delle classi, di nuovo nella loro rappresentazione a stringa:

```
>>> inv_class_mapping = {v: k for k, v in class_mapping.items()}
>>> df['classlabel'] = df['classlabel'].map(inv_class_mapping)
>>> df
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```

Alternativamente, scikit-learn implementa direttamente una comoda classe `LabelEncoder`, che svolge la stessa operazione:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([0, 1, 0])
```

Notate che il metodo `fit_transform` è semplicemente un'abbreviazione delle due chiamate di `fit` e `transform` e possiamo utilizzare il metodo `inverse_transform` per trasformare i numeri interi, corrispondenti alle etichette delle classi, di nuovo nella loro rappresentazione originale a stringa:

```
>>> class_le.inverse_transform(y)
array(['class1', 'class2', 'class1'], dtype=object)
```

Esecuzione di una codifica one-hot su caratteristiche nominali

Nel paragrafo precedente, abbiamo utilizzato un semplice approccio con mappaggio a dizionario per convertire in interi la caratteristica ordinale delle dimensioni. Poiché gli estimatori di scikit-learn trattano le etichette delle classi senza considerare alcun ordine, abbiamo utilizzato la comoda classe `LabelEncoder` per codificare le etichette stringa di nuovo in valori interi. Sembrerebbe possibile utilizzare un approccio simile per trasformare anche la colonna nominale `color` del nostro dataset, come segue:

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

Dopo aver eseguito il codice precedente, la prima colonna dell'array NumPy `X` ora contiene i nuovi valori `color`, codificati nel seguente modo:

- blue → 0

- green → 1
- red → 2

Se ci fermassimo a questo punto e inviassimo l'array al nostro classificatore, commetteremo uno degli errori più comuni nella gestione dei dati categorici. Riuscite a individuare il problema? Anche se i valori relativi al colore non hanno alcun ordine specifico, un algoritmo di apprendimento supporrà ora che *green* sia maggiore di *blue* e che *red* sia maggiore di *green*. Sebbene questa supposizione sia errata, l'algoritmo potrebbe comunque produrre risultati utili. Tuttavia questi risultati non sarebbero ottimali.

Una soluzione comune per questo problema consiste nell'utilizzare una tecnica chiamata *codifica one-hot*. L'idea su cui si basa questo approccio è quella di creare una nuova caratteristica fittizia per ogni valore univoco nella colonna della caratteristica nominale. Qui, convertire la caratteristica `color` in tre nuove caratteristiche: `blue`, `green` e `red`. A questo punto possono essere utilizzati dei valori binari per indicare lo specifico colore di un campione; per esempio, un campione blu può essere codificato come `blue=1`, `green=0`, `red=0`. Per svolgere la trasformazione, potremmo utilizzare lo `OneHotEncoder` implementato nel modulo `scikit-learn.preprocessing`:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> ohe = OneHotEncoder(categorical_features=[0])
>>> ohe.fit_transform(X).toarray()
array([[  0. ,   1. ,   0. ,   1. ,  10.1],
       [  0. ,   0. ,   1. ,   2. ,  13.5],
       [  1. ,   0. ,   0. ,   3. ,  15.3]])
```

Quando abbiamo inizializzato `OneHotEncoder`, abbiamo definito la posizione (la colonna) della variabile che vogliamo trasformare nel parametro `categorical_features` (notate che `color` è la prima colonna nella matrice delle caratteristiche `X`). Per impostazione predefinita, `OneHotEncoder` restituisce una matrice sparsa se utilizziamo il metodo `transform` e convertiamo la rappresentazione a matrice sparsa in un normale array NumPy (*denso*) per scopi di visualizzazione tramite il metodo `toarray`. Le matrici sparse sono semplicemente un modo più efficiente per memorizzare grossi dataset, che tra l'altro è supportato da molte funzioni scikit-learn, il che è particolarmente utile se la matrice contiene una grande quantità di valori a 0. Per omettere il passo `toarray` potremmo inizializzare l'encoder come `OneHotEncoder(…,sparse=False)`, per restituire un normale array NumPy.

Un modo ancora più comodo per creare queste caratteristiche fittizie tramite la codifica one-hot consiste nell'utilizzare il metodo `get_dummies` implementato nei pandas.

Applicato a un DataFrame, il metodo get_dummies convertirà solo le colonne stringa e lascerà intatte tutte le altre colonne:

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
   price  size  color_blue  color_green  color_red
0   10.1     1           0            1          0
1   13.5     2           0            0          1
2   15.3     3           1            0          0
```

Partizionamento di un dataset nei set di addestramento e di test

Abbiamo già introdotto brevemente il concetto di partizionamento di un dataset in set distinti per l'addestramento e il collaudo nel Capitolo 1, *Dare ai computer la capacità di apprendere dai dati* e nel Capitolo 3, *I classificatori di machine learning di scikit-learn*. Ricorderete che il set di test può essere considerato il *test finale* del nostro modello, prima di lasciarlo operare sul mondo reale. In questo paragrafo prepareremo un nuovo dataset, il dataset *Wine*. Dopo aver pre-elaborato il dataset, esploreremo varie tecniche per la selezione delle caratteristiche, con lo scopo di ridurne la dimensionalità.

Wine è un altro dataset open source disponibile nel repository di machine learning UCI (https://archive.ics.uci.edu/ml/datasets/Wine). È costituito da 178 campioni di vini, con 13 caratteristiche che descrivono le loro varie proprietà.

Utilizzando la libreria pandas, leggeremo direttamente il dataset Wine dal repository di machine learning UCI:

```
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash', 'Magnesium',
...                    'Total phenols', 'Flavanoids',
...                    'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>>  print('Class labels', np.unique(df_wine['Class label']))
Class labels [1 2 3]
>>> df_wine.head()
```

Le 13 diverse caratteristiche del dataset *Wine*, che descrivono le proprietà chimiche dei 178 campioni di vini, sono elencate nella Figura 4.3.

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

Figura 4.3

I campioni appartengono a tre diverse classi, 1, 2 e 3, che fanno riferimento ai tre diversi tipi di vitigni che crescono in varie regioni d'Italia.

Un modo comodo per suddividere casualmente questo dataset in un dataset di *test* e un altro di *addestramento* consiste nell'utilizzare la funzione `train_test_split` del modulo `cross_validation` di scikit-learn:

```
>>> from sklearn.cross_validation import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...       train_test_split(X, y, test_size=0.3, random_state=0)
```

Innanzitutto, abbiamo assegnato la rappresentazione ad array NumPy delle colonne delle caratteristiche 1-13 alla variabile `X` e abbiamo assegnato le etichette delle classi (prima colonna) alla variabile `y`. Poi abbiamo utilizzato la funzione `train_test_split` per suddividere casualmente `X` e `y` nei dataset distinti di addestramento e di test. Impostando `test_size=0.3` abbiamo assegnato il 30% dei campioni di vino a `X_test` e `y_test` e il rimanente 70% dei campioni a `X_train` e `y_train`.

NOTA

Se dobbiamo dividere un dataset nei due dataset di addestramento e di test, dobbiamo tenere in considerazione che stiamo sottraendo informazioni preziose, di cui l'algoritmo di apprendimento potrebbe fare buon uso. Pertanto, non vogliamo allocare troppe informazioni nel set di test. Tuttavia, più piccolo è il set di test, più sarà imprecisa la stima dell'errore di generalizzazione. La divisione di un dataset fra i set di addestramento e di test deve valutare questi compromessi. In pratica, le divisioni più comunemente utilizzate sono 60:40, 70:30 o 80:20 a seconda delle dimensioni del dataset iniziale. Tuttavia, per grossi dataset, possono essere comuni e appropriate anche suddivisioni del tipo 90:10 o 99:1. Invece di sbarazzarsi dei dati di test dopo l'addestramento e la valutazione del modello, è sempre opportuno mantenere un classificatore sull'intero dataset, per garantire le migliori prestazioni.

Portare tutte le caratteristiche sulla stessa scala

La scala delle caratteristiche è un elemento fondamentale nella catena di pre-elaborazione e di cui è facile dimenticarsi. Gli alberi decisionali nelle foreste casuali sono fra i pochi algoritmi di machine learning in cui non dobbiamo preoccuparci della scala delle caratteristiche. Tuttavia, la maggior parte degli algoritmi di machine learning e di ottimizzazione si comporta molto meglio se le caratteristiche adottano sulla stessa scala, come abbiamo visto nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, quando abbiamo implementato l'algoritmo di ottimizzazione a discesa del gradiente.

L'importanza della riduzione in scala delle caratteristiche può essere illustrata da un semplice esempio. Supponiamo di avere due caratteristiche, nelle quali una è misurata su una scala da 1 a 10 e la seconda è misurata su una scala da 1 a 100.000. Quando consideriamo la funzione di errore al quadrato di Adaline nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, è facile capire che l'algoritmo sarà occupato principalmente a ottimizzare i pesi sulla base degli errori offerti dalla seconda caratteristica. Un altro esempio è l'algoritmo KNN (*k-nearest neighbors*) con una misurazione della distanza euclidea; le distanze calcolate fra campioni saranno premiate dal secondo asse, ovvero dalla seconda caratteristica.

Ora, vi sono due approcci per portare caratteristiche differenti sulla stessa scala: la *normalizzazione* e la *standardizzazione*. Questi termini vengono frequentemente utilizzati in modo piuttosto vago in vari campi e il loro significato deve essere derivato dal contesto. In generale, la normalizzazione fa riferimento al cambiamento di scala della caratteristica in un intervallo [0, 1], che è un caso speciale di riduzione in scala min-max. Per normalizzare i dati, applichiamo semplicemente la scala min-max a ciascuna colonna delle caratteristiche, dove il nuovo valore $x_{norm}^{(i)}$ di un campione $x^{(i)}$ può essere calcolato nel seguente modo:

Qui, $x^{(i)}$ è un particolare campione, $x_{\min}$ è il più piccolo valore nella colonna della caratteristica e $x_{\max}$ è il valore più grande.

La procedura di riduzione in scala min-max è implementata in scikit-learn e può essere utilizzata nel seguente modo:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

Sebbene la normalizzazione in scala min-max sia una tecnica comunemente utilizzata e che risulta utile quando occorre mantenere i valori in un intervallo ben definito, la standardizzazione può essere più pratica per molti algoritmi di machine learning. Il motivo è che molti modelli lineari, come la regressione logistica e la SVM che abbiamo trattato nel Capitolo 3, *I classificatori di machine learning di scikit-learn*, inizializzano i pesi a 0 o a piccoli valori casuali prossimi a 0. Utilizzando la standardizzazione, centriamo la colonna della caratteristica alla media 0 con deviazione standard 1, in modo che la colonna della caratteristica assuma la forma di una distribuzione normale, dalla quale è più facile derivare i pesi. Inoltre, la standardizzazione mantiene informazioni utili riguardo alle anomalie e rende l'algoritmo meno sensibile a questo problema rispetto alla riduzione in scala min-max, che riporta i dati a un intervallo limitato di valori.

La procedura di standardizzazione può essere espressa dalla seguente equazione:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Qui, μ_x è la media del campione di una determinata colonna di caratteristiche e σ_x è la corrispondente deviazione standard.

La seguente tabella illustra la differenza esistente fra le due tecniche più comunemente utilizzate di riduzione in scala delle caratteristiche, di standardizzazione e di normalizzazione su un semplice dataset campione costituito dai numeri compresi tra 0 e 5:

Input	Standardizzazione	Normalizzazione
0.0	-1.336306	0.0
1.0	-0.801784	0.2
2.0	-0.267261	0.4
3.0	0.267261	0.6
4.0	0.801784	0.8
5.0	1.336306	1.0

Analogamente a `MinMaxScaler`, scikit-learn implementa anche una classe per la standardizzazione:

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

Di nuovo, è importante anche evidenziare il fatto che adattiamo StandardScaler una prima volta sui dati di addestramento e poi utilizziamo questi stessi parametri per trasformare il set di test o ogni nuovo punto dei dati.

Selezione delle caratteristiche appropriate

Se notiamo che un modello si comporta molto meglio su un dataset di addestramento rispetto al dataset di test, questa osservazione è un forte indicatore di *overfitting*. Questo significa che il modello si è adattato con grande precisione alle specifiche osservazioni avvenute nel dataset di addestramento, ma fallisce nella generalizzazione con i dati reali, ovvero il modello ha una *elevata varianza*. Una causa dell'overfitting è il fatto che il nostro modello è troppo complesso per i dati di addestramento forniti; le soluzioni più comuni per ridurre l'errore di generalizzazione possono essere le seguenti:

- raccogliere più dati di addestramento;
- introdurre una penalità per la complessità, tramite la regolarizzazione;
- scegliere un modello più semplice, con un minor numero di parametri;
- ridurre la dimensionalità dei dati.

Raccogliere più dati di addestramento, spesso, non è possibile. Nel prossimo capitolo impareremo una tecnica utile per controllare se sia davvero utile impiegare più dati di addestramento. Nei prossimi paragrafi esamineremo dei metodi comuni per ridurre l'overfitting tramite la regolarizzazione e la riduzione della dimensionalità attraverso la selezione delle caratteristiche.

Soluzioni sparse con la regolarizzazione L1

Ricorderemo dal Capitolo 3, *I classificatori di machine learning di scikit-learn*, che la *regolarizzazione L2* è un approccio che ha lo scopo di ridurre la complessità di un modello, penalizzando i grossi pesi individuali, dove abbiamo definito la norma L2 del nostro vettore dei pesi *w* nel seguente modo:

$$L2: \|\boldsymbol{w}\|_2^2 = \sum_{j=1}^{m} w_j^2$$

Un altro approccio per ridurre la complessità del modello è una versione correlata, ovvero la *regolarizzazione L1*:

$$L1 : \|\boldsymbol{w}\|_1 = \sum_{j=1}^{m} |w_j|$$

Qui, abbiamo semplicemente sostituito alla somma del quadrato dei pesi la somma del valore assoluto dei pesi. Rispetto alla regolarizzazione L2, la regolarizzazione L1 fornisce vettori di caratteristiche sparsi. La maggior parte dei pesi delle caratteristiche sarà uguale a 0. La scarsità dei vettori può essere utile, in pratica, se abbiamo un dataset a elevata dimensionalità contenente molte caratteristiche irrilevanti, specialmente nei casi in cui abbiamo più dimensioni rilevanti che campioni. In questo senso, la regolarizzazione L1 può essere considerata come una tecnica per la selezione delle caratteristiche.

Per comprendere meglio in quale modo la regolarizzazione L1 incoraggia la scarsità, facciamo un passo indietro e diamo un'occhiata all'interpretazione geometrica della regolarizzazione. Tracciamo i contorni di una funzione di costo convessa per due coefficienti di peso w_1 e w_2. Qui, considereremo la somma degli errori al quadrato (*sum of the squared errors – SSE*), la funzione di costo che abbiamo utilizzato per Adaline nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, in quanto è simmetrica e più facile da tracciare rispetto alla funzione di costo della regressione logistica; tuttavia, anche a quest'ultima si applicano gli stessi concetti. Ricordate che il nostro obiettivo è quello di trovare la combinazione dei coefficienti di peso che minimizza la funzione di costo per i dati di addestramento, come illustrato dalla Figura 4.4 (il punto al centro delle ellissi).

Ora, possiamo considerare la regolarizzazione sommando una penalità alla funzione di costo, in modo da incoraggiare l'impiego di pesi più ridotti; in altre parole, penalizziamo i pesi maggiori.

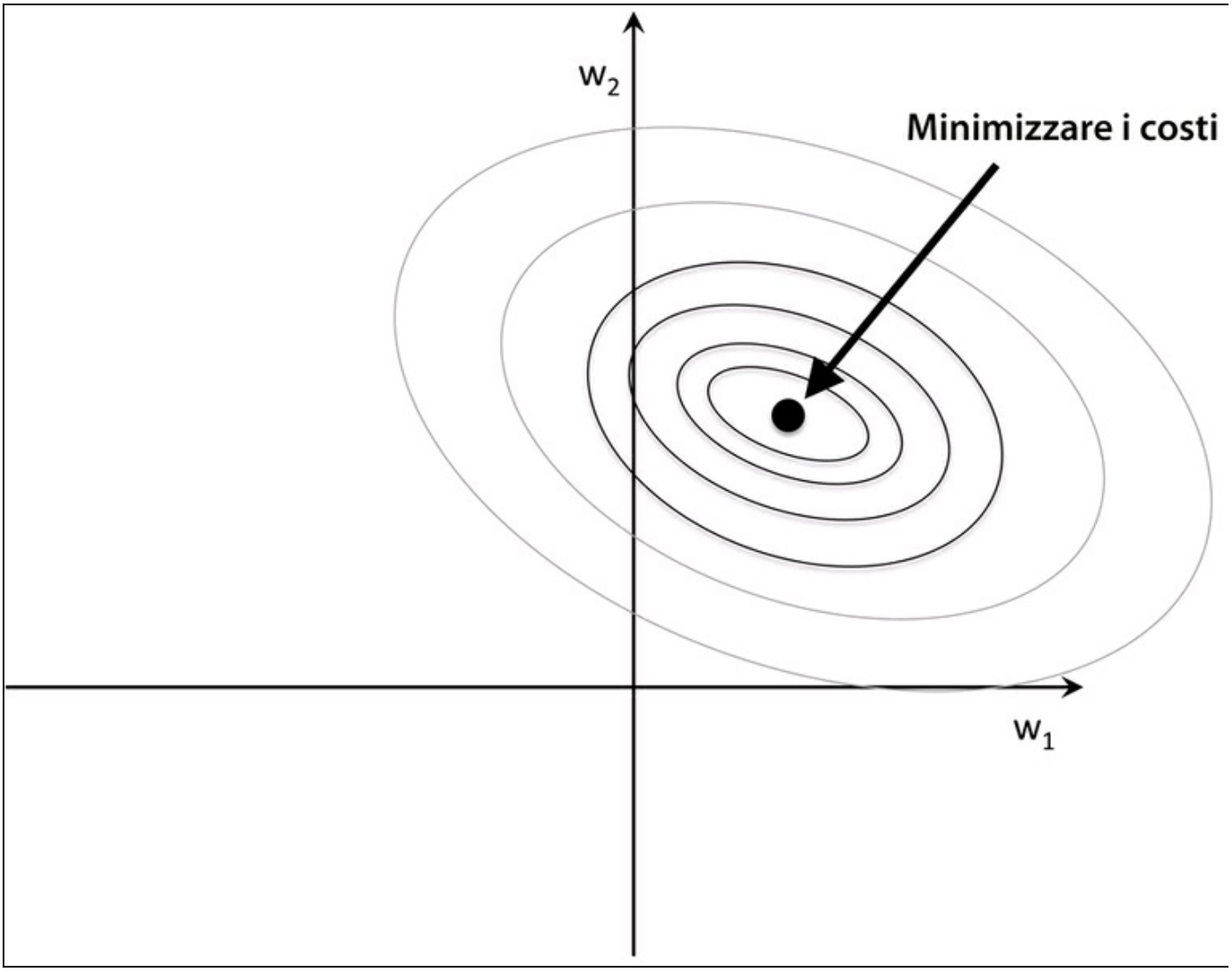


Figura 4.4

Pertanto, incrementando la forza della regolarizzazione tramite il parametro di regolarizzazione λ, riduciamo i pesi verso lo zero e riduciamo la dipendenza del nostro modello dai dati di addestramento. Possiamo illustrare questo concetto nella Figura 4.5 per la penalità L2.

Il termine di regolarizzazione quadratica L2 viene rappresentato da un cerchio grigio. Qui, i nostri coefficienti di peso non possono superare il nostro *budget* di regolarizzazione (la combinazione dei coefficienti di peso non può fuoriuscire dall'area grigia). D'altra parte, vogliamo comunque minimizzare la funzione di costo. Sotto il vincolo della penalità, il nostro tentativo migliore consiste nello scegliere il punto in cui il cerchio L2 interseca i contorni della funzione di costo non penalizzata. Maggiore è il valore del parametro di regolarizzazione λ, più velocemente cresce la funzione penalizzata dei costi, il che porta ad avere un cerchio L2 più ridotto. Per esempio, se incrementiamo il parametro di regolarizzazione verso l'infinito, i coefficienti di peso diventeranno in realtà 0, si

collocheranno al centro della sfera del cerchio L2. Per riepilogare la sostanza di questo esempio, il nostro scopo è quello di minimizzare la somma della funzione di costo non penalizzata e del termine di penalità, il che significa sommare il bias e preferire un modello più semplice per ridurre la varianza, in assenza di dati di addestramento sufficienti per adattare il modello.

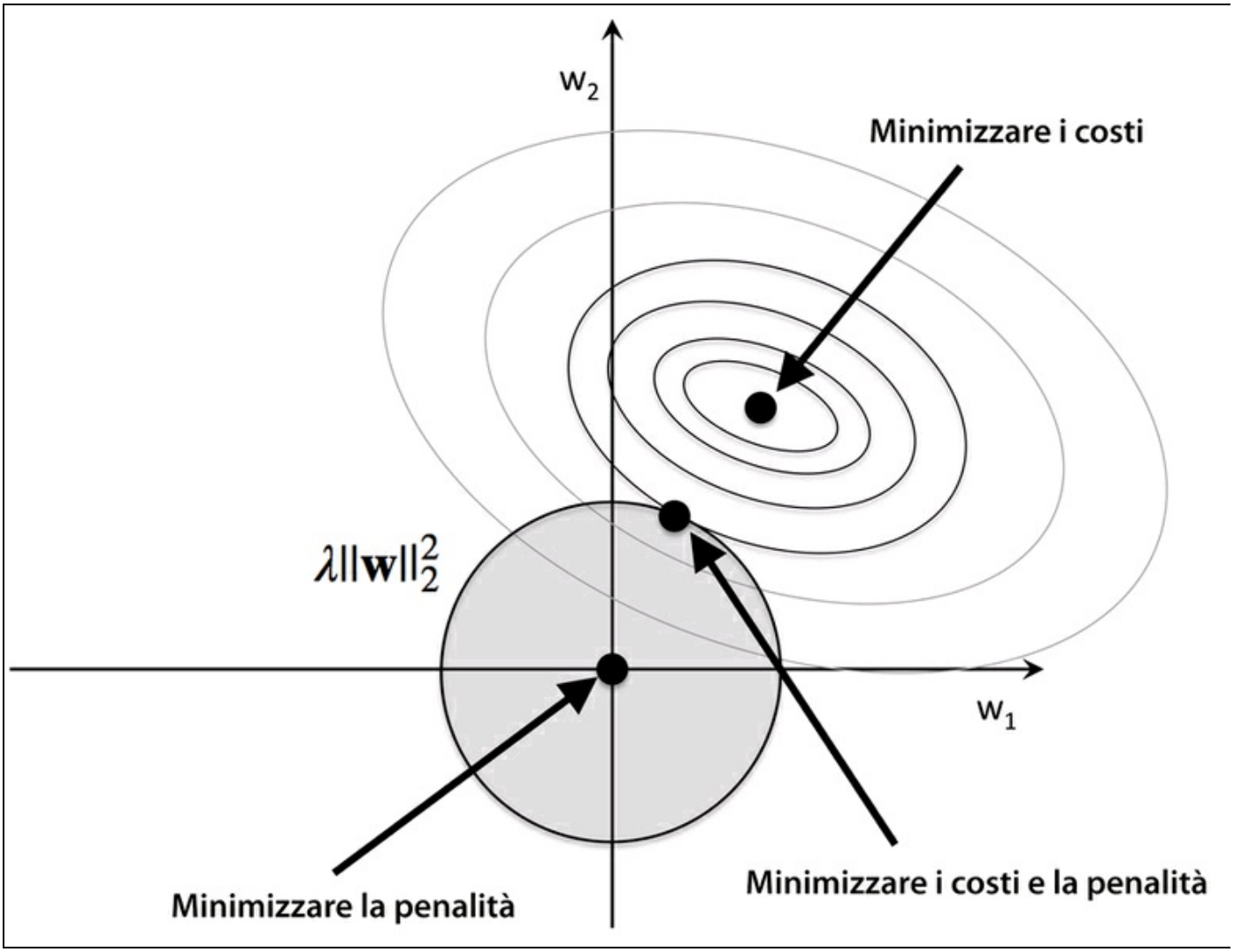


Figura 4.5

Ora parliamo della regolarizzazione L1 e della sparsità. Il concetto di base fra la regolarizzazione L1 è simile a quello di cui abbiamo appena parlato. Tuttavia, poiché la penalità L1 è la somma del valore assoluto dei coefficienti di peso (ricordate che il termine L2 è quadratico), possiamo rappresentarla come un *budget* romboidale, come illustrato nella Figura 4.6.

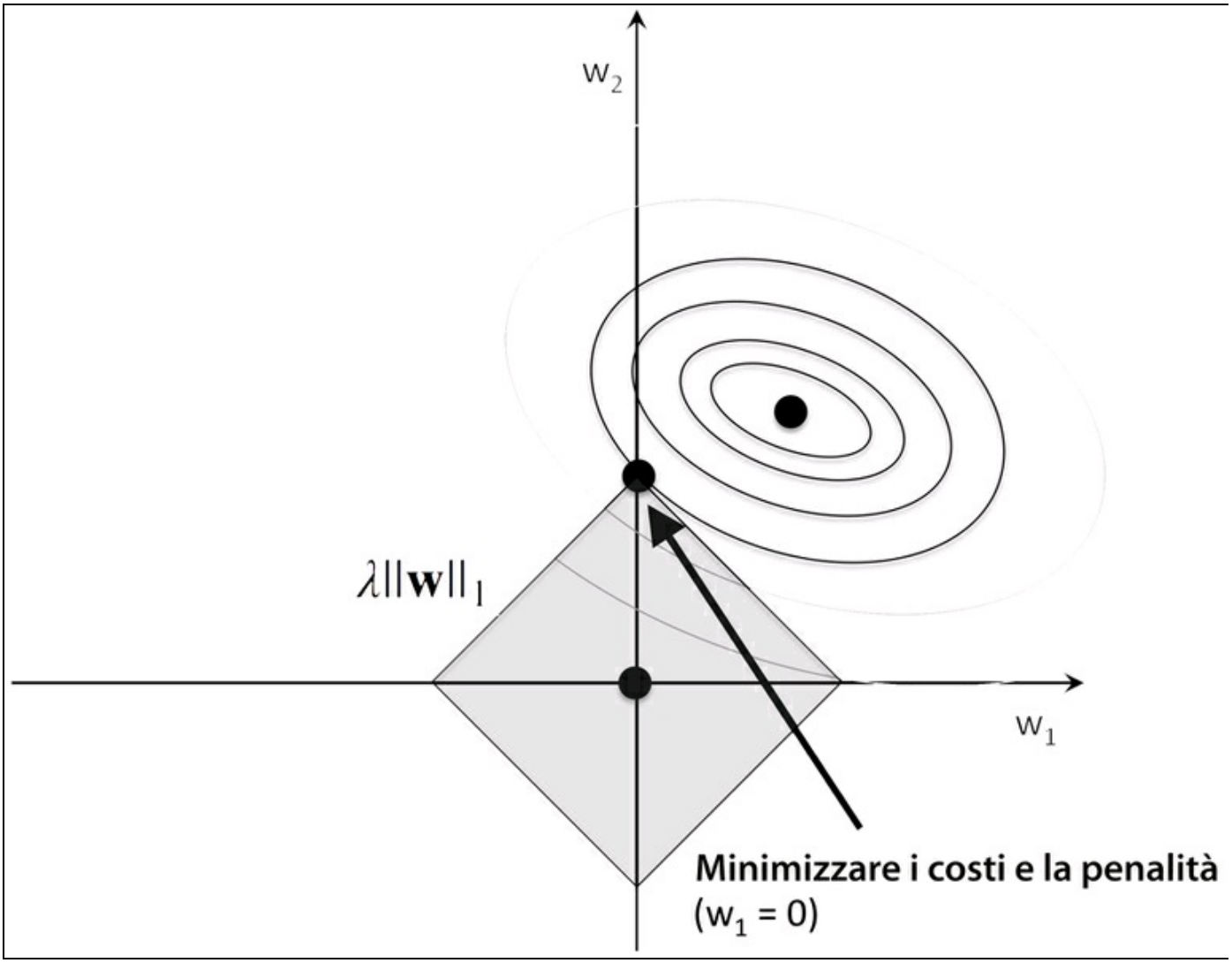


Figura 4.6

Nella figura, possiamo vedere che il contorno della funzione di costo tocca il rombo L1 nel punto $w_1 = 0$. Poiché i contorni di un sistema regolarizzato L1 sono netti, è più probabile che l'ottimo, ovvero l'intersezione fra le ellissi della funzione di costo e i confini del rombo L1 sia situato sugli assi, il che incoraggia la sparsità. I dettagli matematici del motivo per cui la regolarizzazione L1 può portare a soluzioni sparse non rientrano negli scopi di questo libro. Se siete interessati, potete trovare un paragrafo eccellente di confronto fra regolarizzazione L2 e L1 nel paragrafo 3.4 del libro *The Elements of Statistical Learning*, di Trevor Hastie, Robert Tibshirani e Jerome Friedman, Springer.

Per i modelli regolarizzati in scikit-learn che supportano la regolarizzazione L1, possiamo semplicemente impostare il parametro `penalty` a `'l1'` per ottenere la soluzione sparsa:

```
>>> from sklearn.linear_model import LogisticRegression
>>> LogisticRegression(penalty='l1')
```

Applicata ai dati standardizzati del dataset Wine, la regressione logistica regolarizzata L1 fornirà la seguente soluzione sparsa:

```
>>> lr = LogisticRegression(penalty='l1', C=0.1)
>>> lr.fit(X_train_std, y_train)
>>> print('Training accuracy:', lr.score(X_train_std, y_train))
Training accuracy: 0.983870967742
>>> print('Test accuracy:', lr.score(X_test_std, y_test))
Test accuracy: 0.981481481481
```

La precisione dell'addestramento e nei test (entrambe al 98%) non rilevano alcun problema di overfitting del modello. Quando accediamo ai termini di intercettazione tramite l'attributo `lr.intercept_`, possiamo vedere che l'array restituisce tre valori:

```
>>> lr.intercept_
array([-0.38379237, -0.1580855 , -0.70047966])
```

Poiché adattiamo l'oggetto `LogisticRegression` su un dataset multiclasse, questo utilizza l'approccio *One-vs-Rest* (*OvR*) standard, dove la prima intercettazione appartiene al modello adattato alla classe 1 rispetto alle classi 2 e 3; il secondo valore è l'intercettazione del modello adattato alla classe 2 rispetto alle classi 1 e 3 e il terzo valore è l'intercettazione del modello adattato alla classe 3 rispetto alle classi 1 e 2:

```
>>> lr.coef_
array([[ 0.280, 0.000, 0.000, -0.0282, 0.000,
         0.000, 0.710, 0.000, 0.000, 0.000,
         0.000, 0.000, 1.236],
       [-0.644, -0.0688 , -0.0572, 0.000, 0.000,
         0.000, 0.000, 0.000, 0.000, -0.927,
         0.060, 0.000, -0.371],
       [ 0.000, 0.061, 0.000, 0.000, 0.000,
         0.000, -0.637, 0.000, 0.000, 0.499,
        -0.358, -0.570, 0.000
       ]])
```

L'array dei pesi, cui si accede tramite l'attributo `lr.coef_`, contiene tre righe di coefficienti di peso, un vettore dei pesi per ciascuna classe. Ogni riga è costituita da 13 pesi, dove ciascun peso viene moltiplicato per la rispettiva caratteristica nel dataset Wine a 13 dimensioni, per calcolare l'input della rete:

$$z = w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^T \boldsymbol{x}$$

Notiamo che i vettori peso sono sparsi, ovvero hanno solo alcuni elementi diversi da 0. Come risultato della regolarizzazione L1, che serve quale metodo per la selezione delle caratteristiche, abbiamo appena addestrato un modello che è resistente nei confronti delle caratteristiche del dataset che risultano essere potenzialmente irrilevanti.

Infine tracciamo il percorso di regolarizzazione, che è costituito dai coefficienti di peso delle varie caratteristiche per le varie intensità di regolarizzazione:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> colors = ['blue', 'green', 'red', 'cyan',
...         'magenta', 'yellow', 'black',
...         'pink', 'lightgreen', 'lightblue',
...         'gray', 'indigo', 'orange']
>>> weights, params = [], []
>>> for c in np.arange(-4, 6):
...     lr = LogisticRegression(penalty='l1',
...                             C=10**c,
...                             random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> for column, color in zip(range(weights.shape[1]), colors):
...     plt.plot(params, weights[:, column],
...              label=df_wine.columns[column+1],
...              color=color)
>>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>>> plt.xlim([10**(-5), 10**5])
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.xscale('log')
>>> plt.legend(loc='upper left')
>>> ax.legend(loc='upper center',
...           bbox_to_anchor=(1.38, 1.03),
...           ncol=1, fancybox=True)
>>> plt.show()
```

Il grafico risultante (Figura 4.7) ci fornisce ulteriori conoscenze sul comportamento della regolarizzazione L1. Come possiamo vedere, tutti i pesi delle caratteristiche saranno a 0 se penalizziamo il modello con un parametro di regolarizzazione "forte" ($C < 0.1$); C è l'inverso del parametro di regolarizzazione λ.

Algoritmi sequenziali per la selezione delle caratteristiche

Un modo alternativo per ridurre la complessità del modello ed evitare il problema dell'overfitting è la *riduzione della dimensionalità* tramite la selezione delle caratteristiche, tecnica particolarmente utile per i modelli non regolarizzati. Vi sono due categorie principali di tecniche per la riduzione della dimensionalità: *selezione della caratteristica* ed *estrazione della caratteristica*.

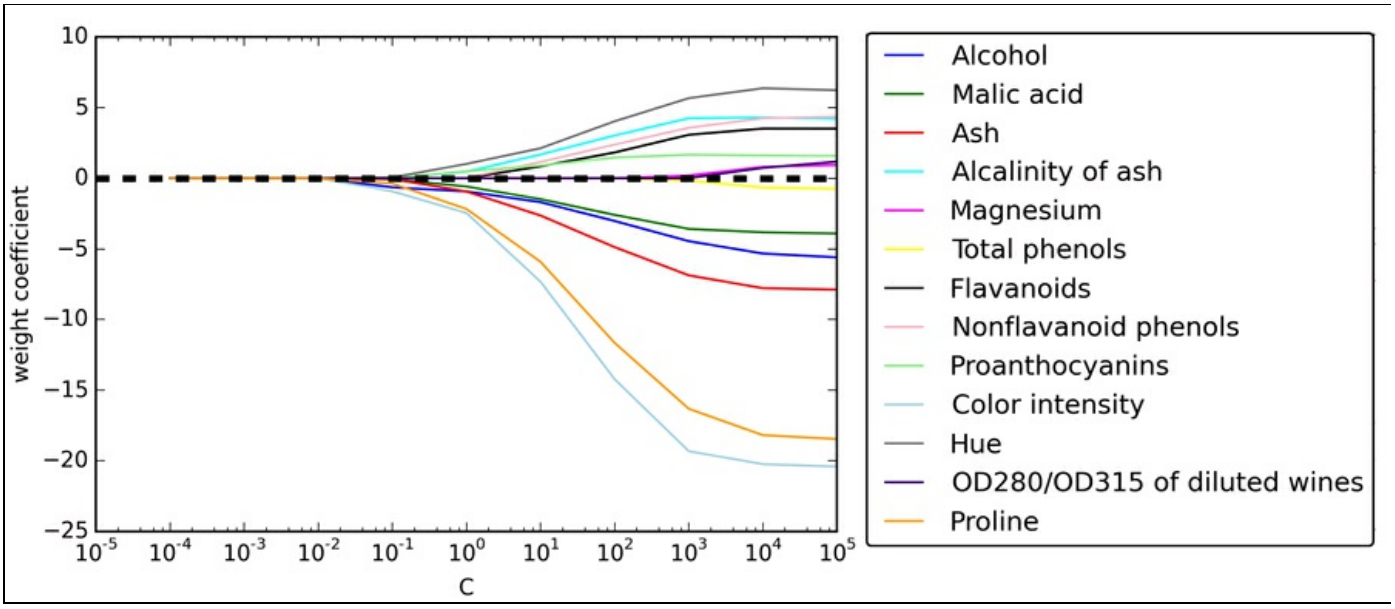


Figura 4.7

Utilizzando la selezione della caratteristica, selezioniamo un sottoinsieme delle caratteristiche originarie. Nell'estrazione della caratteristica deriviamo le informazioni dall'insieme di caratteristiche, per costruire un nuovo sottospazio di caratteristiche. In questo paragrafo esamineremo una classica famiglia di algoritmi per la selezione delle caratteristiche. Nel Capitolo 5, *Compressione dei dati tramite la riduzione della dimensionalità*, impareremo a usare varie tecniche di estrazione delle caratteristiche per comprimere un dataset in un sottospazio di caratteristiche a minore dimensionalità.

Gli algoritmi sequenziali per la selezione delle caratteristiche sono una famiglia di algoritmi di ricerca *greedy*, utilizzati per ridurre uno spazio di caratteristiche *d-dimensionale* in un sottospazio di caratteristiche *k-dimensionale* dove $k < d$. La motivazione su cui si basano gli algoritmi per la selezione delle caratteristiche consiste nel selezionare automaticamente un sottoinsieme delle caratteristiche che sia particolarmente rilevante per il problema, in modo da migliorare l'efficienza computazionale o ridurre l'errore di generalizzazione del modello, rimuovendo le caratteristiche irrilevanti o il rumore, il che può essere utile per gli algoritmi che non supportano la regolarizzazione. Un classico algoritmo sequenziale di selezione delle caratteristiche è la *selezione sequenziale all'indietro* o *Sequential Backward Selection* (*SBS*), che mira a ridurre la dimensionalità del sottospazio di caratteristiche iniziale con un minimo decadimento in termini prestazionali del classificatore, in modo da migliorare l'efficienza computazionale globale. In alcuni

casi, SBS può perfino migliorare la capacità predittiva del modello, qualora tale modello soffra di overfitting.

NOTA

Gli algoritmi greedy ("aggressivi") eseguono scelte ottimali a livello locale in ogni fase di un problema di ricerca combinatoria e generalmente forniscono una soluzione subottimale al problema, al contrario degli algoritmi di ricerca esaustivi, che valutano tutte le possibili combinazioni e garantiscono di trovare la soluzione ottimale. Tuttavia, in pratica, una ricerca esaustiva non è fattibile dal punto di vista computazionale, mentre gli algoritmi greedy forniscono una soluzione meno complessa e, computazionalmente, più efficiente.

L'idea su cui si basa l'algoritmo SBS è piuttosto semplice: SBS rimuove sequenzialmente l'intero insieme delle caratteristiche finché il nuovo sottospazio delle caratteristiche contiene solo il numero desiderato di caratteristiche. Per determinare quale caratteristiche devono essere rimosse in ogni fase, dobbiamo definire la funzione criterio J che vogliamo minimizzare. Il criterio calcolato dalla funzione può essere semplicemente la differenza prestazionale del classificatore prima e dopo la rimozione di una determinata caratteristica. Poi la caratteristica da eliminare in ciascuna fase può semplicemente essere definita come la caratteristica che massimizza questo criterio; in termini più intuitivi, a ogni fase eliminiamo la caratteristica che provoca il minore degrado prestazionale dopo la rimozione. Sulla base della definizione precedente di SBS, possiamo descrivere l'algoritmo in quattro semplici fasi.

1. Inizializzare l'algoritmo con $k = d$, dove d è la dimensionalità dell'intero spazio delle caratteristiche $\mathbf{X}_d$.
2. Determinare la caratteristica x^- che massimizza il criterio $x^- = argmax J\left(\mathbf{X}_k - x\right)$, dove $x \in \mathbf{X}_k$.
3. Rimuovere la caratteristica x^- dall'insieme di caratteristiche: $\mathbf{X}_k - 1 = \mathbf{X}_k - x^-, k = k - 1$.
4. Uscire se k è uguale al numero di caratteristiche desiderate, altrimenti tornare al Passo 2.

NOTA

Per una valutazione dettagliata di vari algoritmi sequenziali operano sulle caratteristiche, consultate *Comparative Study of Techniques for Large Scale Feature Selection*, F. Ferri, P. Pudil, M. Hatef e J. Kittler, in "Comparative study of techniques for large-scale feature selection. Pattern Recognition in Practice IV", pagine 403–413, 1994.

Sfortunatamente, l'algoritmo SBS non è ancora implementato in scikit-learn. Ma poiché è così semplice, possiamo occuparci di implementarlo direttamente in

Python:

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.cross_validation import train_test_split
from sklearn.metrics import accuracy_score
class SBS():
    def __init__(self, estimator, k_features,
        scoring=accuracy_score,
        test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)

        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state

    def fit(self, X, y):
        X_train, X_test, y_train, y_test = \
            train_test_split(X, y, test_size=self.test_size,
                             random_state=self.random_state)
        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                 X_test, y_test, self.indices_)
        self.scores_ = [score]

        while dim > self.k_features:
            scores = []
            subsets = []
            for p in combinations(self.indices_, r=dim-1):
                score = self._calc_score(X_train, y_train,
                                         X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]

        return self

    def transform(self, X):
        return X[:, self.indices_]
    def _calc_score(self, X_train, y_train,
                    X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score
```

Nell'implementazione precedente, abbiamo definito il parametro k_features per specificare il numero di caratteristiche che vogliamo ottenere. Per default, utilizziamo accuracy_score di scikit-learn per valutare le prestazioni di un modello e quale sistema di stima per la classificazione sul sottoinsieme delle caratteristiche. All'interno del ciclo while del metodo fit, i sottoinsiemi di caratteristiche creati dalla funzione itertools.combination vengono valutati e ridotti finché il sottoinsieme delle caratteristiche non ha la dimensionalità desiderata. A ogni iterazione, il punteggio di accuratezza del miglior sottoinsieme viene raccolto in una lista self.scores_, sulla base

di un dataset di test `X_test` creato internamente. Utilizzeremo questi punteggi successivamente per valutare i risultati. Gli indici di colonna del sottoinsieme di caratteristiche finale vengono assegnati a `self.indices_`, che possiamo utilizzare con il metodo `transform` per ottenere un nuovo array di dati con le colonne di caratteristiche selezionate. Notate che, invece di calcolare esplicitamente il criterio all'interno del metodo `fit`, abbiamo semplicemente rimosso la caratteristica che non è contenuta nel sottoinsieme di caratteristiche che offre le prestazioni migliori.

Ora vediamo in azione la nostra implementazione di SBS utilizzando il classificatore KNN di scikit-learn:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> import matplotlib.pyplot as plt
>>> knn = KNeighborsClassifier(n_neighbors=2)
>>> sbs = SBS(knn, k_features=1)
>>> sbs.fit(X_train_std, y_train)
```

Sebbene la nostra implementazione di SBS suddivida già il dataset in un dataset di test e un altro di addestramento all'interno della funzione `fit`, forniamo comunque all'algoritmo il dataset di addestramento `X_train`. Il metodo SBS `fit` creerà poi i nuovi sottoinsiemi di addestramento per il collaudo (convalida) e l'addestramento, ragione per cui questo test è anche chiamato *dataset di convalida*. *Questo approccio è necessario per impedire che il nostro set di test originale entri a far parte dei dati di addestramento.*

Ricordate che il nostro algoritmo SBS raccoglie in ciascuna fase, i punteggi del miglior sottoinsieme di caratteristiche. Pertanto procediamo con la parte più interessante dell'implementazione e tracciamo l'accuratezza del classificatore KNN, che è stata calcolata sul dataset di convalida. Il codice è il seguente:

```
>>> k_feat = [len(k) for k in sbs.subsets_]
>>> plt.plot(k_feat, sbs.scores_, marker='o')
>>> plt.ylim([0.7, 1.1])
>>> plt.ylabel('Accuracy')
>>> plt.xlabel('Number of features')
>>> plt.grid()
>>> plt.show()
```

Come possiamo vedere dalla Figura 4.8, l'accuratezza del classificatore KNN è migliorata sul dataset di convalida quando abbiamo ridotto il numero di caratteristiche, il che è probabilmente dovuto proprio alla riduzione della dimensionalità di cui abbiamo parlato nel contesto dell'algoritmo KNN nel Capitolo 3, *I classificatori di machine learning di scikit-learn*. Inoltre, possiamo vedere nel grafico che il classificatore ha ottenuto una precisione del 100% per *k={5, 6, 7, 8, 9, 10}*.

Per curiosità, vediamo quali sono le cinque caratteristiche che hanno fornito risultati prestazionali così buoni sul dataset di convalida:

```
>>> k5 = list(sbs.subsets_[8])
>>> print(df_wine.columns[1:][k5])
Index(['Alcohol', 'Malic acid', 'Alcalinity of ash', 'Hue', 'Proline'], dtype='object')
```

Utilizzando il codice precedente, abbiamo ottenuto gli indici delle colonne del sottoinsieme delle cinque caratteristiche dalla nona posizione dell'attributo `sbs.subsets_` e abbiamo restituito i nomi delle caratteristiche corrispondenti tratti dall'indice colonna del `DataFrame` del pandas di Wine.

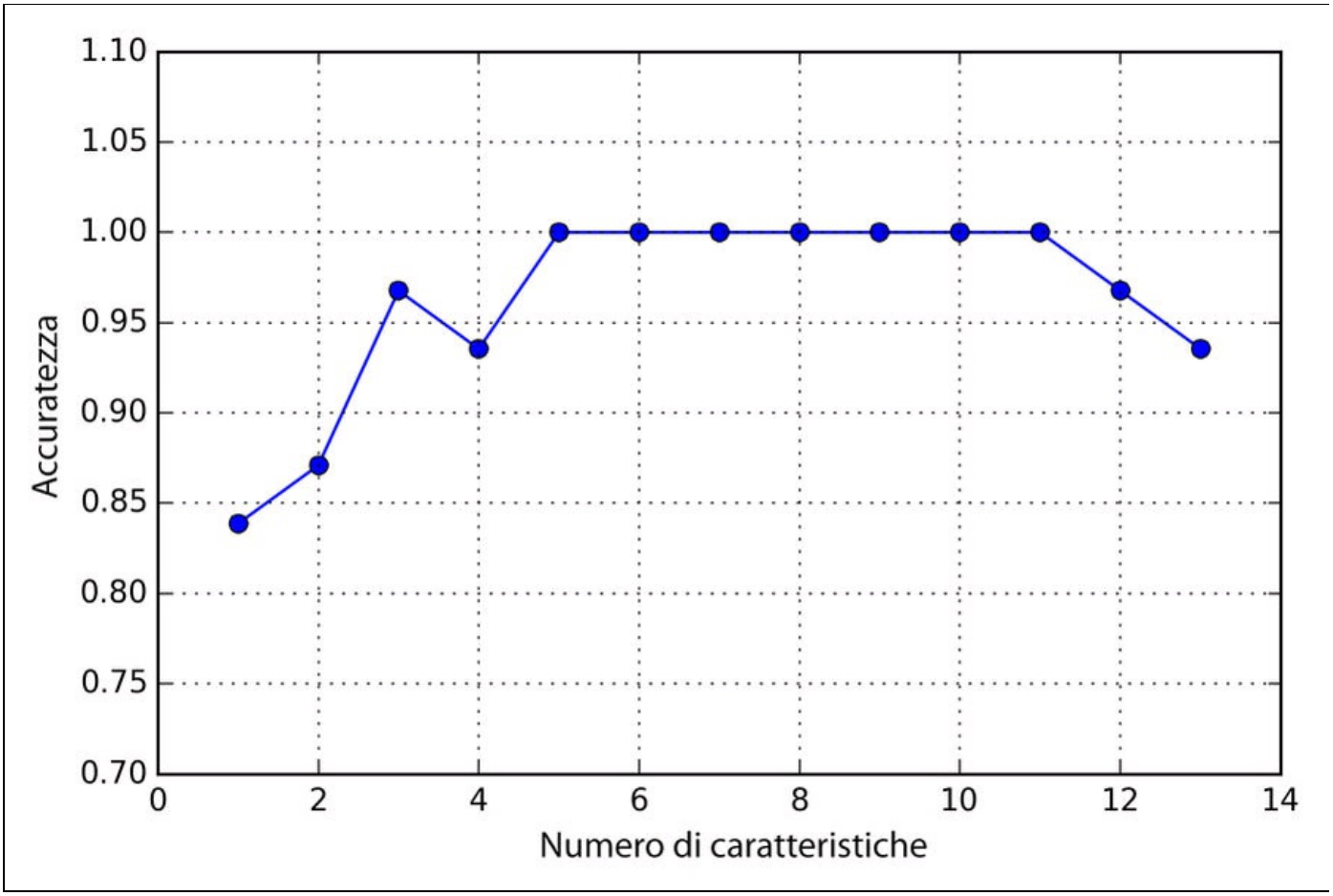


Figura 4.8

Ora valutiamo le prestazioni del classificatore KNN sul set di test originario:

```
>>> knn.fit(X_train_std, y_train)
>>> print('Training accuracy:', knn.score(X_train_std, y_train))
Training accuracy: 0.983870967742
>>> print('Test accuracy:', knn.score(X_test_std, y_test))
Test accuracy: 0.944444444444
```

Nel codice precedente, abbiamo utilizzato l'intero insieme di caratteristiche e abbiamo ottenuto circa il 98.4% di precisione sul dataset di addestramento. Tuttavia, l'accuratezza sul dataset di test è stata leggermente inferiore (circa il 94.4%), il che indica un leggero overfitting. Ora utilizziamo il sottoinsieme delle cinque caratteristiche selezionate per vedere la qualità delle prestazioni di KNN:

```
>>> knn.fit(X_train_std[:, k5], y_train)
>>> print('Training accuracy:',
...       knn.score(X_train_std[:, k5], y_train))
Training accuracy: 0.959677419355
>>> print('Test accuracy:',
...       knn.score(X_test_std[:, k5], y_test))
Test accuracy: 0.962962962963
```

Utilizzando meno della metà delle caratteristiche originali del dataset Wine, l'accuratezza della previsione sul set di test è migliorata di quasi il 2%. Inoltre, abbiamo ridotto l'overfitting, il che si deduce dalla piccola differenza esistente fra l'accuratezza del test (circa il 96. 3%) e dell'addestramento (96. 0%).

NOTA

Scikit-learn non fornisce molti altri algoritmi per la selezione delle caratteristiche. Tra questi vi è l'eliminazione a ritroso ricorsiva, che si basa sui pesi delle caratteristiche, i metodi basati su alberi per selezionare le caratteristiche per importanza e i test statistici uninvariati. Un'ampia discussione dei vari metodi per la selezione delle caratteristiche non può rientrare negli scopi di questo libro, ma un buon riepilogo ricco di esempi si può trovare in: http://scikitlearn.org/stable/modules/feature_selection.html.

Valutazione dell'importanza delle caratteristiche con le foreste casuali

Nei paragrafi precedenti, abbiamo imparato a utilizzare la regolarizzazione L1 per azzerare le caratteristiche irrilevanti tramite la regressione logistica e abbiamo utilizzato l'algoritmo SBS per la selezione delle caratteristiche. Un altro approccio utile per selezionare le caratteristiche rilevanti da un dataset consiste nell'utilizzare una foresta casuale, una tecnica d'insieme che abbiamo introdotto nel Capitolo 3, *I classificatori di machine learning di scikit-learn*. Utilizzando una foresta casuale, possiamo misurare l'importanza delle caratteristiche come la riduzione media delle impurità, calcolata da tutti gli alberi decisionali della foresta senza effettuare alcuna supposizione sul fatto che i dati siano separabili linearmente oppure no. Fortunatamente, l'implementazione della foresta casuale di scikit-learn raccoglie già l'importanza delle caratteristiche per noi e quindi possiamo accedervi tramite l'attributo `feature_importances_` dopo aver impiegato un `RandomForestClassifier`. Tramite il seguente ciclo, addestreremo una foresta di 10.000 alberi sul dataset Wine e valuteremo e classificheremo le 13 caratteristiche sulla base delle rispettive misure di importanza. Ricordate (dalla nostra discussione nel Capitolo 3, *I classificatori di machine learning di scikit-learn*) che non dobbiamo utilizzare modelli basati su alberi standardizzati o normalizzati. Il codice è il seguente:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> feat_labels = df_wine.columns[1:]
>>> forest = RandomForestClassifier(n_estimators=10000,
...                                 random_state=0,
...                                 n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[indices[f]],
...                             importances[indices[f]]))
1) Color intensity              0.182483
2) Proline                      0.158610
3) Flavanoids                   0.150948
4) OD280/OD315 of diluted wines   0.131987
5) Alcohol                      0.106589
6) Hue                          0.078243
7) Total phenols                0.060718
8) Alcalinity of ash            0.032033
9) Malic acid                   0.025400
10) Proanthocyanins             0.022351
11) Magnesium                   0.022078
12) Nonflavanoid phenols        0.014645
13) Ash                         0.013916
>>> plt.title('Feature Importances')
>>> plt.bar(range(X_train.shape[1]),
...         importances[indices],
...         color='lightblue',
...         align='center')
>>> plt.xticks(range(X_train.shape[1]),
```

```
...         feat_labels[indices], rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()
```

Dopo aver eseguito il codice precedente, abbiamo creato un grafico che elenca le diverse caratteristiche del dataset Wine in ordine di importanza relativa (Figura 4.9); notate che le importanze delle caratteristiche sono normalizzate, in modo da produrre una somma pari a 1.0.

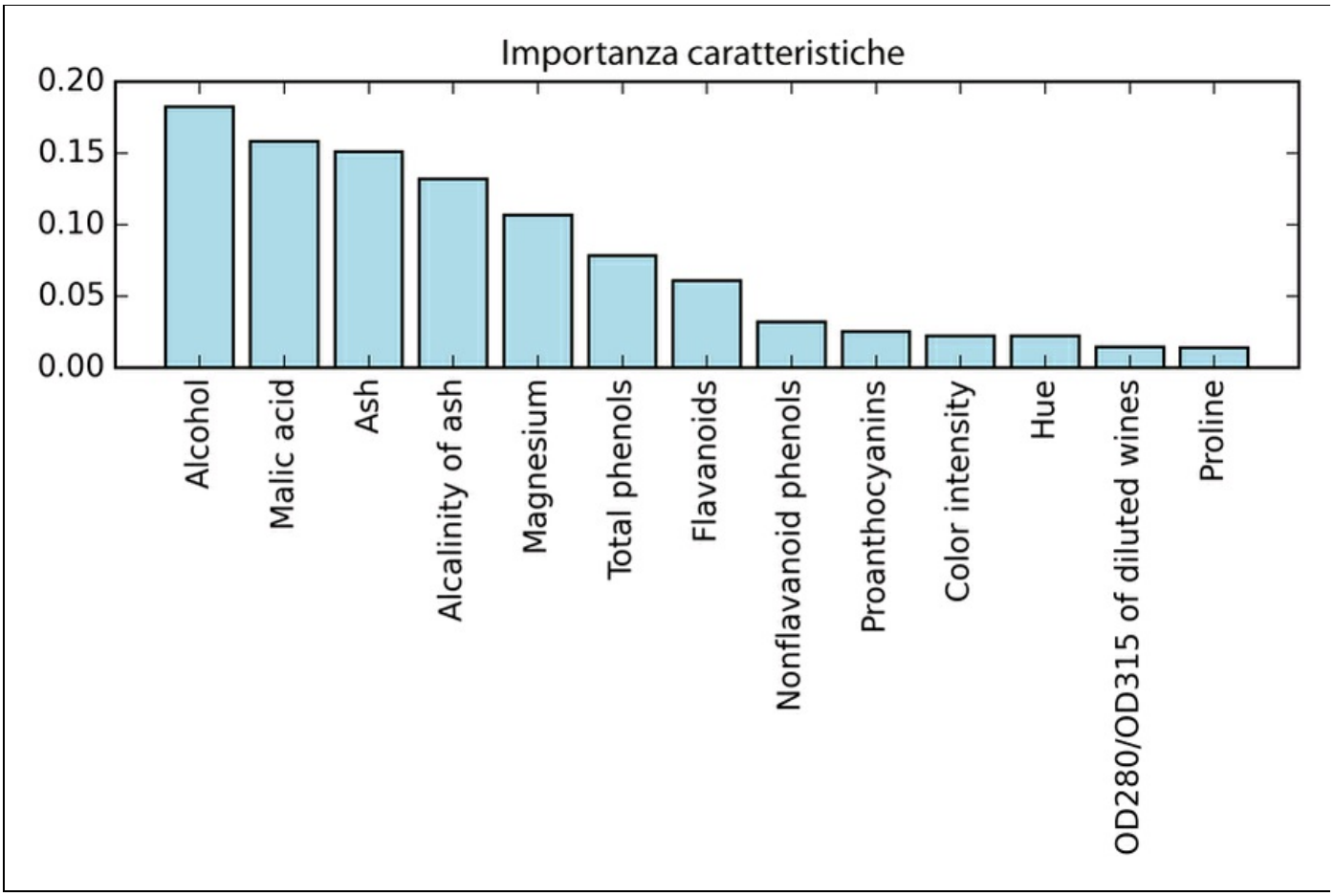


Figura 4.9

Possiamo concludere che il contenuto di alcol del vino è la caratteristica più discriminativa nel dataset sulla base del decremento medio delle impurità nei 10.000 alberi decisionali. È interessante notare che le tre caratteristiche che hanno ricevuto la valutazione più elevata nel grafico precedente sono anche le prime cinque caratteristiche scelte dall'algoritmo SBS che abbiamo implementato nel paragrafo precedente. Tuttavia, per quanto riguarda l'interpretabilità, la tecnica della foresta casuale offre un *indizio* importante che vale la pena di menzionare. Per esempio, se due o più caratteristiche sono molto correlate fra loro, una caratteristica può essere valutata come molto elevata mentre le informazioni dell'altra caratteristica (o delle altre caratteristiche) potrebbe non essere catturata appieno.

D’altro canto, non vogliamo preoccuparci di questo problema se siamo interessati semplicemente alle prestazioni di previsione di un modello piuttosto che all’interpretazione dell’importanza delle caratteristiche. Per concludere questo paragrafo sull’importanza delle caratteristiche e le foreste casuali, vale la pena di menzionare che scikit-learn implementa anche un metodo `transform` che seleziona le caratteristiche sulla base di una soglia specificata dall’utente dopo l’addestramento del modello, soluzione utile se vogliamo utilizzare `RandomForestClassifier` come selettore delle caratteristiche e passo intermedio in una catena scikit-learn, che ci consente di connettere più passi di pre-elaborazione con un estimatore, come vedremo nel Capitolo 6, *Valutazione dei modelli e ottimizzazione degli iperparametri*. Per esempio, potremmo impostare una soglia pari a 0.15 per ridurre il dataset alle tre caratteristiche più importanti, *Alcohol*, *Malic acid* e *Ash*, utilizzando il codice seguente:

```
>>> X_selected = forest.transform(X_train, threshold=0.15)
>>> X_selected.shape
(124, 3)
```

Riepilogo

Abbiamo iniziato questo capitolo osservando le tecniche utili per assicurarci di gestire correttamente i dati mancanti. Prima di inviare i dati a un algoritmo di machine learning, dobbiamo anche assicurarci di codificare correttamente le variabili delle categorie e abbiamo visto come possiamo mappare le caratteristiche ordinali e nominali su rappresentazioni intere.

Inoltre, abbiamo trattato brevemente la regolarizzazione L1, che può aiutarci a evitare il problema di overfitting, riducendo la complessità di un modello. Come approccio alternativo per la rimozione delle caratteristiche irrilevanti, abbiamo utilizzato un algoritmo sequenziale per selezionare le caratteristiche significative da un dataset.

Nel prossimo capitolo tratteremo un altro approccio utile per la riduzione della dimensionalità: l'estrazione delle caratteristiche. Questo ci consentirà di comprimere le caratteristiche in un sottospazio di dimensioni inferiori, invece di eliminarle completamente come avviene nel caso della selezione.

Compressione dei dati tramite la riduzione della dimensionalità

Nel Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione*, abbiamo parlato dei vari approcci per la riduzione della dimensionalità di un dataset utilizzando varie tecniche di selezione delle caratteristiche. Un approccio alternativo alla selezione delle caratteristiche, per ridurne la dimensionalità, è l'*estrazione delle caratteristiche*. In questo capitolo parleremo di tre tecniche fondamentali che ci aiuteranno a condensare il contenuto delle informazioni di un dataset, trasformandolo in un nuovo sottospazio di caratteristiche di dimensionalità inferiore rispetto a quello originale. La compressione dei dati è un argomento importante nell'ambito del machine learning e ci aiuterà a memorizzare e analizzare la quantità crescente di dati che vengono prodotti raccolti al giorno d'oggi. In questo capitolo tratteremo i seguenti argomenti.

- Analisi *Principal Component* (*PCA*) per la compressione dei dati senza supervisione.
- Analisi *Linear Discriminant* (*LDA*) quale tecnica di riduzione della dimensionalità con supervisione per massimizzare la separabilità delle classi.
- Riduzione della dimensionalità non lineare tramite l'analisi *Kernel Principal Component* (*KPCA*).

Riduzione della dimensionalità senza supervisione tramite l'analisi del componente principale (PCA)

Come abbiamo già visto con la selezione delle caratteristiche, possiamo utilizzare *l'estrazione* delle caratteristiche, con lo scopo di ridurre il loro numero nel dataset. Tuttavia, mentre con gli algoritmi di selezione delle caratteristiche, come nel caso della *selezione sequenziale all'indietro* (*SBS*) abbiamo mantenuto le caratteristiche originali, utilizziamo l'estrazione delle caratteristiche per trasformare o proiettare i dati in un nuovo spazio di caratteristiche. Nel contesto della riduzione della dimensionalità, l'estrazione delle caratteristiche può essere considerata come un approccio a compressione dei dati, con l'obiettivo di conservare la maggior parte delle informazioni rilevanti. L'estrazione delle caratteristiche viene tipicamente utilizzata per migliorare l'efficienza computazionale, ma può anche proteggere contro la *maledizione della dimensionalità*, specialmente quando si opera su modelli non regolarizzati.

L'analisi *principal component* (*PCA*) è una tecnica di trasformazione lineare senza supervisione che è ampiamente utilizzata in vari campi, principalmente per la riduzione della dimensionalità. Altre applicazioni tipiche dell'analisi PCA comprendono l'analisi esplorativa dei dati e l'eliminazione del rumore nel mercato delle azioni e nell'analisi dei dati del genoma e ancora nei livelli di espressione del genoma nel campo della bioinformatica. L'analisi PCA ci aiuta a identificare gli schemi presenti nei dati, sulla base della correlazione fra le caratteristiche. In estrema sintesi, l'analisi PCA mira a trovare le direzioni di massima varianza all'interno di dati a elevata dimensionalità, per poi proiettarli in un nuovo sottospazio avente dimensioni uguali o inferiori rispetto all'originale. Gli assi ortogonali (componenti principali) del nuovo sottospazio possono essere interpretati come le direzioni di massima varianza sulla base del vincolo che i nuovi assi delle caratteristiche siano ortogonali fra loro, come illustrato nella Figura 5.1. Qui, x_1 e x_2 sono gli assi originali delle caratteristiche e *PC1* e *PC2* sono i componenti principali.

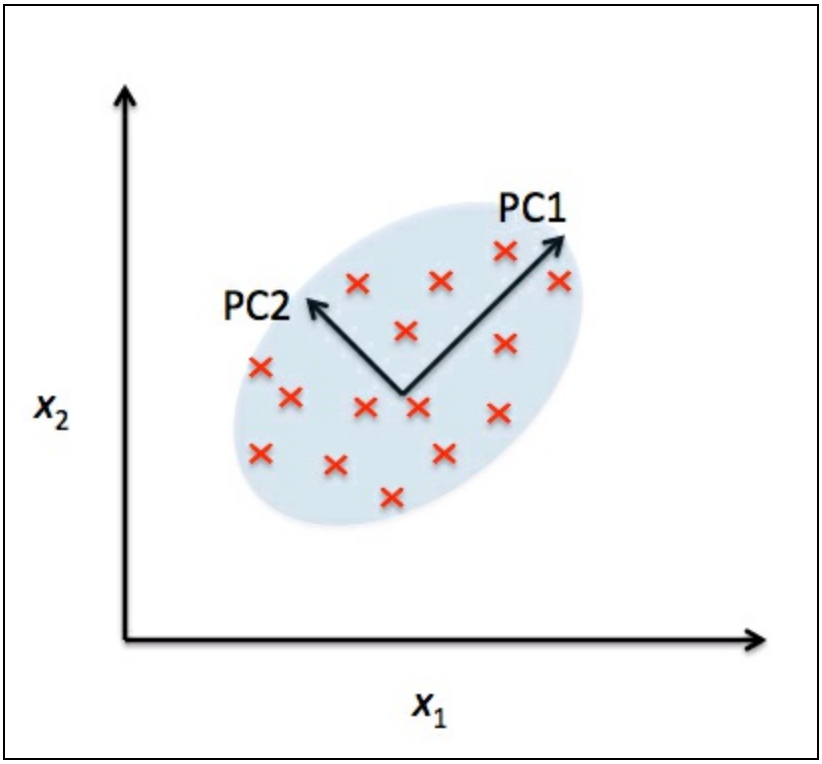


Figura 5.1

Se utilizziamo l'analisi PCA per la riduzione della dimensionalità, costruiamo una matrice di trasformazione $d \times k$-dimensionale di nome $\boldsymbol{W}$, che ci consente di mappare un vettore campione $\boldsymbol{x}$ in un nuovo sottospazio k-dimensionale delle caratteristiche che ha dimensioni inferiori rispetto allo spazio di caratteristiche originale, d-dimensionale:

$$\boldsymbol{x} = \left[x_1, x_2, \ldots, x_d\right], \quad \boldsymbol{x} \in \mathbb{R}^d$$

$$\downarrow \boldsymbol{x}\boldsymbol{W}, \quad \boldsymbol{W} \in \mathbb{R}^{d \times k}$$

$$\mathbf{z} = \left[z_1, z_2, \ldots, z_k\right], \quad \mathbf{z} \in \mathbb{R}^k$$

Come risultato della trasformazione dei dati d-dimensionali originali in questo nuovo sottospazio k-dimensionale (tipicamente $k << d$), il primo componente principale avrà la massima varianza possibile e tutti i successivi componenti principali avranno la massima varianza possibile sulla base del fatto che non sono correlati (in quanto sono ortogonali) agli altri componenti principali. Notate che le direzioni PCA sono molto sensibili alla riduzione in scala dei dati e dobbiamo standardizzare le caratteristiche *prima* dell'analisi PCA se tali caratteristiche sono

state misurate su scale differenti e intendiamo assegnare la stessa importanza a tutte le caratteristiche.

Prima di osservare in maggiore derraglio l'algoritmo PCA per la riduzione della dimensionalità, riepiloghiamo l'approccio, condensandolo in alcuni semplici passi.

1. Standardizzare il dataset d-dimensionale.
2. Costruire la matrice di covarianza.
3. Decomporre la matrice di covarianza nei suoi autovettori e autovalori.
4. Selezionare k autovettori vettori che corrispondano ai k più grandi autovalori, dove k è la dimensionalità del nuovo sottospazio delle caratteristiche ($k \leq d$).
5. Costruire una matrice di proiezione w dai k autovettori "superiori".
6. Trasformare il dataset di input d-dimensionale x utilizzando la matrice di proiezione w per ottenere il nuovo sottospazio delle caratteristiche, k-dimensionale.

Varianza totale e spiegata

In questo paragrafo ci occuperemo dei primi quattro passi di un'analisi del componente principale: la standardizzazione dei dati, la costruzione della matrice di covarianza, l'ottenimento degli autovalori e degli autovettori della matrice di covarianza e l'ordinamento degli autovalori riducendo l'ordine per valutare gli autovettori.

Innanzitutto, inizieremo caricando il dataset *Wine* su cui abbiamo già lavorato nel Capitolo 4,*Costruire buoni set di addestramento: la pre-elaborazione*:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data', header=None)
```

Poi elaboreremo i dati *Wine* per produrre set distinti di addestramento e di test (rispettivamente pari al 70% e al 30% dei dati) e li standardizzaremo in modo che abbiano varianza unitaria.

```
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...             train_test_split(X, y,
...             test_size=0.3, random_state=0)
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Dopo aver completato i passi obbligatori di pre-elaborazione eseguendo il codice precedente, passiamo al secondo passo: costruire la matrice di covarianza. La matrice di covarianza è simmetrica $d \times d$-dimensionale, dove d è il numero delle

dimensioni del dataset, e conserva le covarianza a coppie fra le varie caratteristiche. Per esempio, la covarianza fra le due caratteristiche x_j e x_k sul livello della popolazione può essere calcolata tramite la seguente equazione:

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

Qui, μ_j e μ_k sono le medie del campione, rispettivamente della caratteristica j e k. Notate che, se standardizzate il dataset, le medie del campione sono pari a 0. Una covarianza positiva fra due caratteristiche indica che esse aumentano o si riducono insieme, mentre una covarianza negativa indica il fatto che le caratteristiche variano, ma in direzioni opposte. Per esempio, una matrice di covarianza di tre caratteristiche può essere scritta come (notate che il simbolo Σ sta per la lettera greca sigma, che non deve essere confusa con il simbolo di *sommatoria*):

$$\sum = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

Gli autovettori della matrice di covarianza rappresentano i componenti principali (le direzioni di massima varianza), mentre i corrispondenti autovalori definiranno la loro ampiezza. Nel caso del dataset *Wine*, otterremo 13 autovettori e autovalori dalla matrice di covarianza 13×13-dimensionale.

Ora otteniamo le autocoppie della matrice di covarianza. Come ricorderete sicuramente dalle lezioni di algebra, un autovalore $\boldsymbol{v}$ soddisfa la seguente condizione:

$$\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$$

Qui, λ è uno scalare: l'autovalore. Poiché il calcolo manuale degli autovettori e degli autovalori è un'operazione noiosa e complessa, utilizzeremo la funzione linalg.eig di NumPy per ottenere le autocoppie della matrice di covarianza di *Wine*:

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nEigenvalues \n%s' % eigen_vals)
Eigenvalues
[ 4.8923083   2.46635032  1.42809973  1.01233462  0.84906459  0.60181514
0.52251546  0.08414846  0.33051429  0.29595018  0.16831254  0.21432212
0.2399553 ]
```

Utilizzando la funzione numpy.cov, abbiamo calcolato la matrice di covarianza del dataset di addestramento standardizzato. Utilizzando la funzione linalg.eig, abbiamo eseguito l'autodecomposizione che ha fornito un vettore (eigen_vals) costituito da 13 autovalori e i corrispondenti autovettori conservati come colonne in una matrice 13×13-dimensionale (eigen_vecs).

NOTA

Anche se la funzione numpy.linalg.eig è stata progettata con lo scopo di decomporre matrici quadrate non simmetriche, scoprirete che in alcuni casi restituisce autovalori complessi. Una funzione correlata, numpy.linalg.eigh, ha lo scopo di decomporre matrici hermetiane, il che è numericamente un approccio più stabile per lavorare con le matrici simmetriche come la matrice di covarianza; numpy.linalg.eigh fornisce già veri autovalori.

Poiché vogliamo ridurre la dimensionalità del nostro dataset comprimendolo in un nuovo sottospazio di caratteristiche, selezioneremo solo il sottoinsieme degli autovettori (componenti principali) che contengono la maggior parte delle informazioni (varianza). Poiché gli autovalori definiscono l'ordine di grandezza degli autovettori, dobbiamo ordinare gli autovalori in senso decrescente; siamo interessati ai k autovettori superiori sulla base dei valori dei corrispondenti autovalori. Ma prima di raccogliere questi k autovettori più informativi, tracciamo i *rapporti della varianza spiegata* degli autovalori.

Il rapporto della varianza spiegata dell'autovalore λ_j è semplicemente il rapporto fra l'autovalore λ_j e la somma totale degli autovalori:

$$\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$$

Utilizzando la funzione NumPy cumsum, possiamo quindi calcolare la somma cumulativa delle varianza spiegata, che tracceremo tramite la funzione step di matplotlib:

```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...            sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)
>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, alpha=0.5, align='center',
...         label='individual explained variance')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...         label='cumulative explained variance')
>>> plt.ylabel('Explained variance ratio')
>>> plt.xlabel('Principal components')
>>> plt.legend(loc='best')
>>> plt.show()
```

Il grafico risultante (Figura 5.2) indica che, da solo, il primo componente principale conta per il 40% della varianza. Inoltre, possiamo vedere che i primi due componenti principali, combinati, spiegano quasi il 60% della varianza presente nei dati.

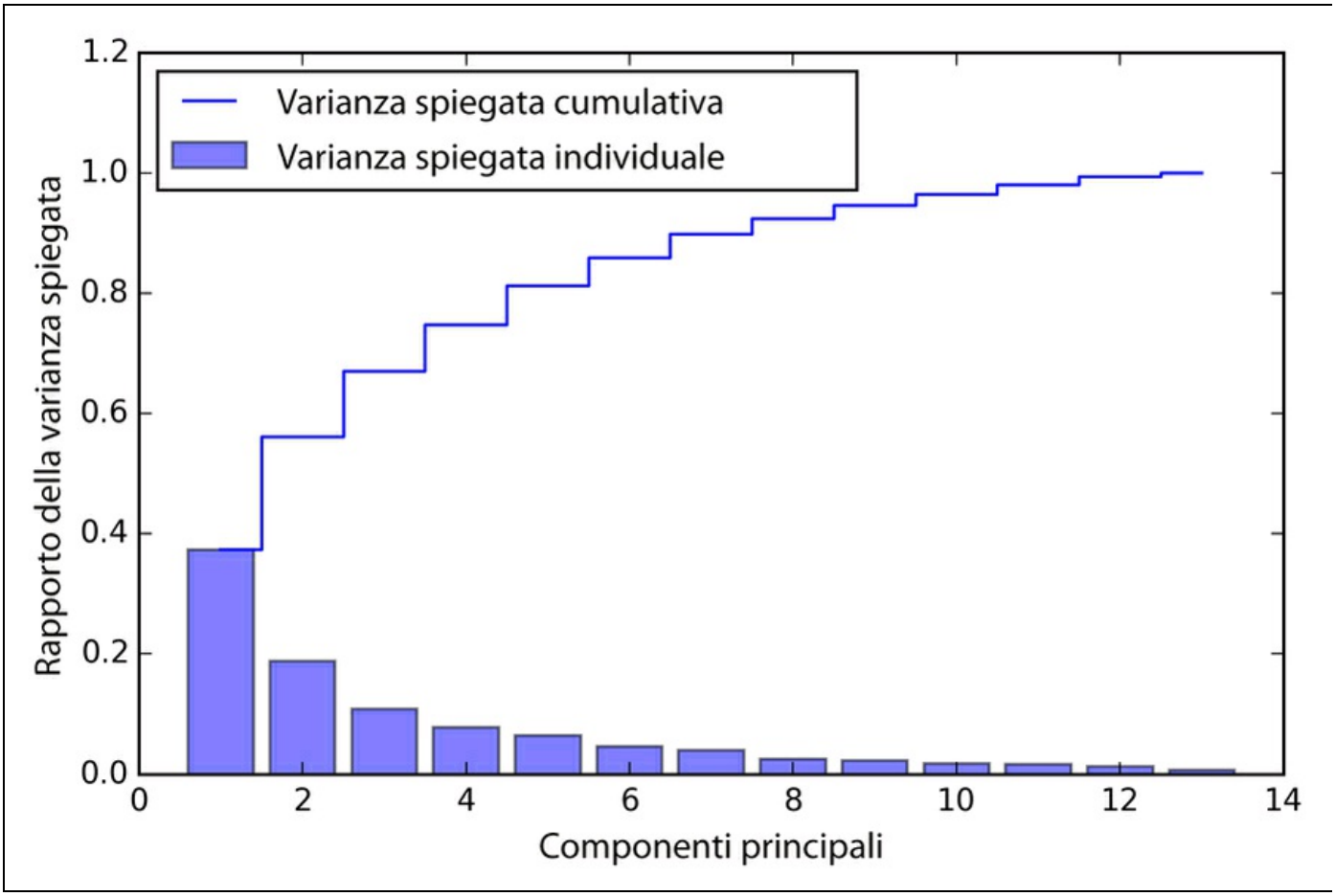


Figura 5.2

Sebbene il grafico della varianza spiegata ci parli dell'importanza della caratteristiche che abbiamo calcolato nel Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione* utilizzando le foreste casuali, dobbiamo ricordarci che l'analisi PCA è un metodo senza supervisione, il che significa che le informazioni relative alle etichette delle classi vengono ignorate. Mentre una foresta casuale utilizza le informazioni sull'appartenenza alla classe per calcolare le impurità del nodo, la varianza misura la dispersione dei valori lungo l'asse della caratteristica.

Trasformazione di una caratteristica

Dopo aver decomposto con successo la matrice di covarianza nelle autocoppie, procediamo con gli ultimi tre passi di trasformazione del dataset *Wine* sui nuovi assi

del componente principale. In questo paragrafo, ordineremo le autocoppie in ordine discendente degli autovalori, costruiremo una matrice di proiezione dai autovettori selezionati e utilizzeremo la matrice di proiezione per trasformare i dati in un sottospazio di dimensioni inferiori.

Partiamo ordinando le autocoppie riducendo l'ordine degli autovalori:

```
>>> eigen_pairs =[(np.abs(eigen_vals[i]),eigen_vecs[:,i])
...               for i inrange(len(eigen_vals))]
>>> eigen_pairs.sort(reverse=True)
```

Poi raccogliamo i due autovettori che corrispondono ai due valori maggiori, per catturare circa il 60% della varianza presente in questo dataset. Badate che scegliamo solo due autovettori solo per scopi illustrativi, in quanto, più avanti in questo paragrafo, tracceremo graficamente i dati tramite due grafici bidimensionali a dispersione. Nella pratica, il numero dei componenti principali deve essere determinato da un compromesso fra efficienza computazionale e prestazioni del classificatore:

```
>>> w= np.hstack((eigen_pairs[0][1][:, np.newaxis],
...               eigen_pairs[1][1][:, np.newaxis]))
>>> print('Matrix W:\n',w)
Matrix W:
[[ 0.14669811  0.50417079]
[-0.24224554  0.24216889]
[-0.02993442  0.28698484]
[-0.25519002 -0.06468718]
[ 0.12079772  0.22995385]
[ 0.38934455  0.09363991]
[ 0.42326486  0.01088622]
[-0.30634956  0.01870216]
[ 0.30572219  0.03040352]
[-0.09869191  0.54527081]
[ 0.30032535 -0.27924322]
[ 0.36821154 -0.174365  ]
[ 0.29259713  0.36315461]]
```

Tramite il codice precedente, abbiamo creato una matrice di proiezione 13×2-dimensionale W dai due autovalori superiori. Utilizzando la matrice di proiezione, possiamo trasformare un campione x (rappresentato come un vettore riga 1×13-dimensionale) nel sottospazio PCA, ottenendo x', un vettore campione che ora è bidimensionale ed è costituito da due nuove caratteristiche:

$$x' = xW$$

```
>>> X_train_std[0].dot(w)
array([ 2.59891628,  0.00484089])
```

Analogamente, possiamo trasformare l'intero dataset di addestramento 124×13-dimensionale nei due principali componenti, calcolando il prodotto della matrice:

$$X' = XW$$

```
>>> X_train_pca = X_train_std.dot(w)
```

Infine, visualizzeremo il set di addestramento *Wine* trasformato, che ora è conservato come una matrice a 124×2 dimensioni in un grafico bidimensionale a dispersione:

```
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=l, marker=m)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Come possiamo vedere nel grafico rappresentato nella Figura 5.3, i dati sono più dispersi lungo l'asse *x* (il primo componente principale) che lungo l'asse *y* (il secondo componente principale) il che è coerente con il grafico del rapporto della varianza spiegata che abbiamo creato nel paragrafo precedente. Tuttavia, intuitivamente, possiamo vedere che un classificatore lineare sarà comunque in grado di separare bene le classi.

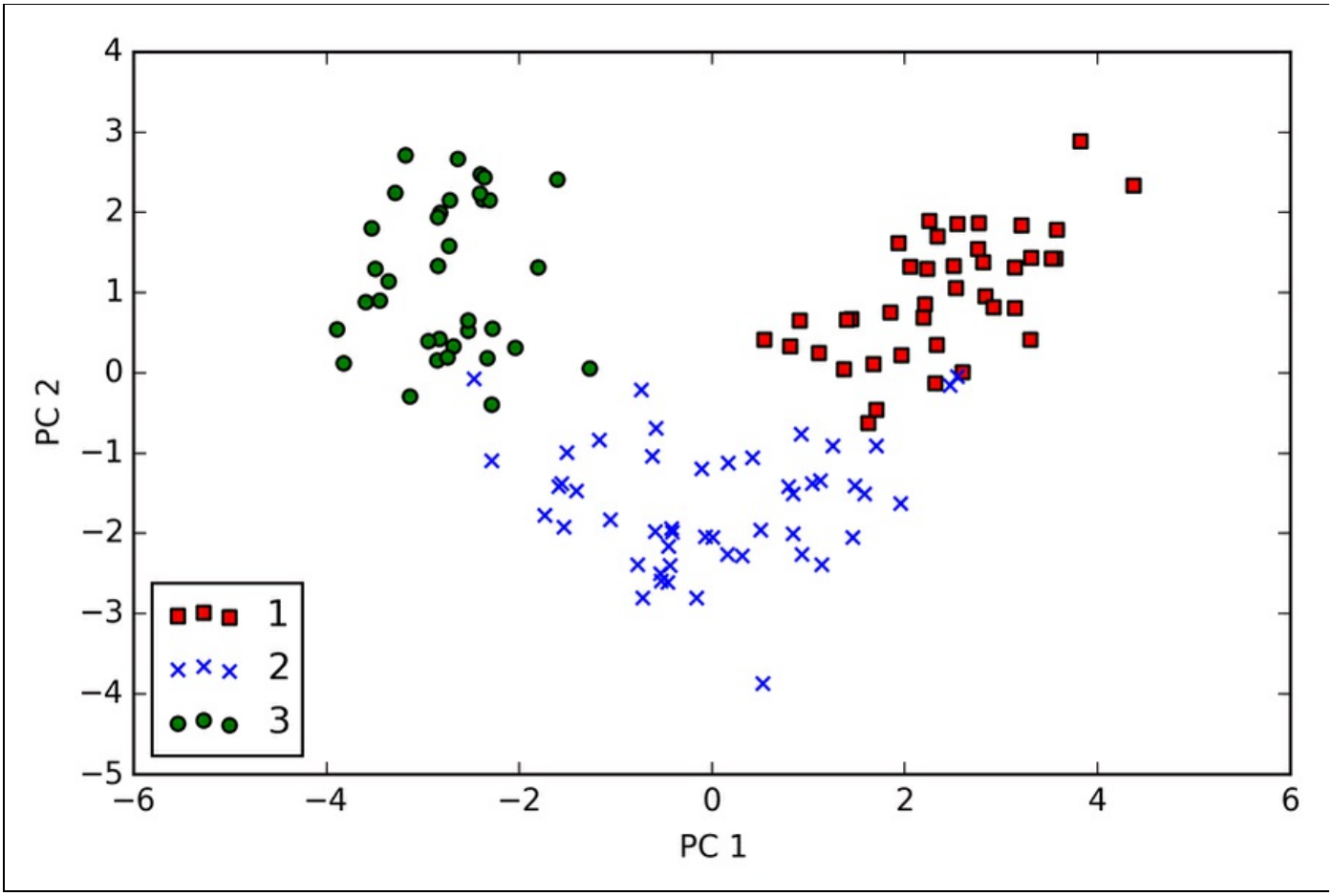


Figura 5.3

Sebbene abbiamo codificato le informazioni delle etichette delle classi con lo scopo di illustrarle in un grafico a dispersione, dobbiamo sempre considerare che

l'analisi PCA è una tecnica senza supervisione, che non utilizza informazioni sulle etichette delle classi.

L'analisi del componente principale in scikit-learn

L'approccio piuttosto prolisso del paragrafo precedente dovrebbe averci aiutato a seguire il funzionamento interno dell'analisi PCA. Ora tratteremo l'uso della classe `PCA` implementata in scikit-learn. `PCA` è un'altra delle classi di trasformazione di scikit-learn, dove prima adattiamo il modello utilizzando i dati di addestramento e poi trasformiamo i dati di addestramento e i dati di test utilizzando gli stessi parametri del modello. Ora, utilizziamo la classe `PCA` di scikit-learn applicandola al dataset di addestramento di *Wine*, classifichiamo i campioni trasformati tramite la regressione logistica e visualizziamo le regioni decisionali tramite la funzione `plot_decision_region` che abbiamo definito nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*:

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
# setup marker generator and color map
markers = ('s', 'x', 'o', '^', 'v')
colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
cmap = ListedColormap(colors[:len(np.unique(y))])
# plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)
```

```
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression()
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Tramite il codice precedente, dovremmo vedere le regioni decisionali per il modello di addestramento ridotto ai due assi dei componenti principali (Figura 5.4).

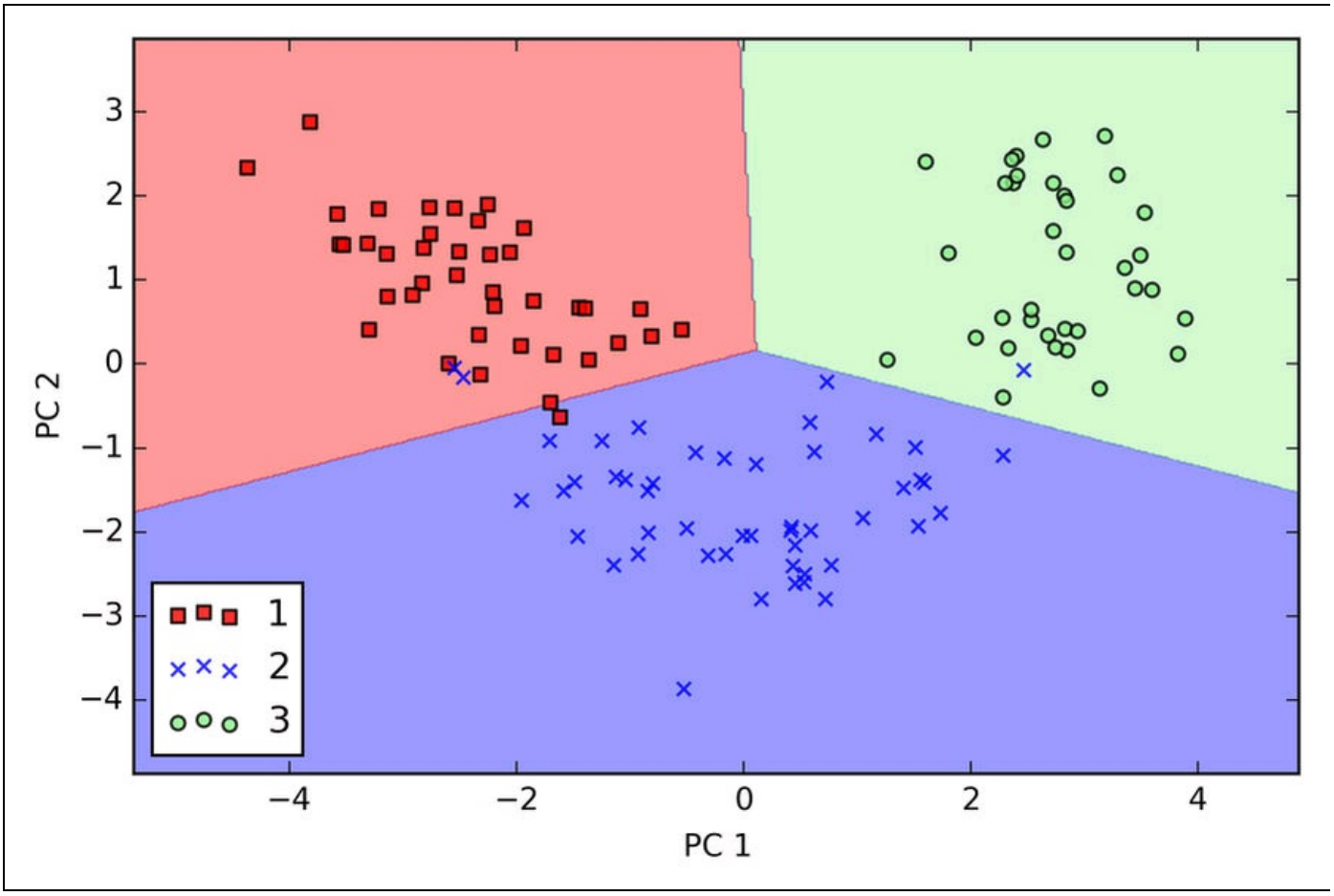


Figura 5.4

Se confrontiamo la proiezione PCA di scikit-learn con la nostra implementazione di PCA, notiamo che quest'ultimo grafico è un'immagine speculare del precedente grafico PCA ottenuto tramite il nostro approccio a passi. Notate che questo non è dovuto a un errore di una delle due implementazione: il motivo di questa differenza è il fatto che, a seconda dell'autosolutore, gli autovettori possono avere segno negativo o positivo. Non che questo conti davvero, ma potremmo dover semplicemente invertire l'immagine speculare moltiplicando i dati per -1, se vogliamo; notate che gli autovettori sono tipicamente scalati alla lunghezza unitaria 1. Per completezza, tracciamo le regioni decisionali della regressione logistica sul dataset di test trasformato, per vedere se è in grado di separare bene le classi:

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Dopo aver tracciato in un grafico (Figura 5.5) le regioni decisionali per il set di test eseguendo il codice precedente, possiamo vedere che la regressione logistica si comporta piuttosto bene su questo piccolo sottospazio bidimensionale delle caratteristiche e sbaglia a classificare un solo campione del dataset di test.

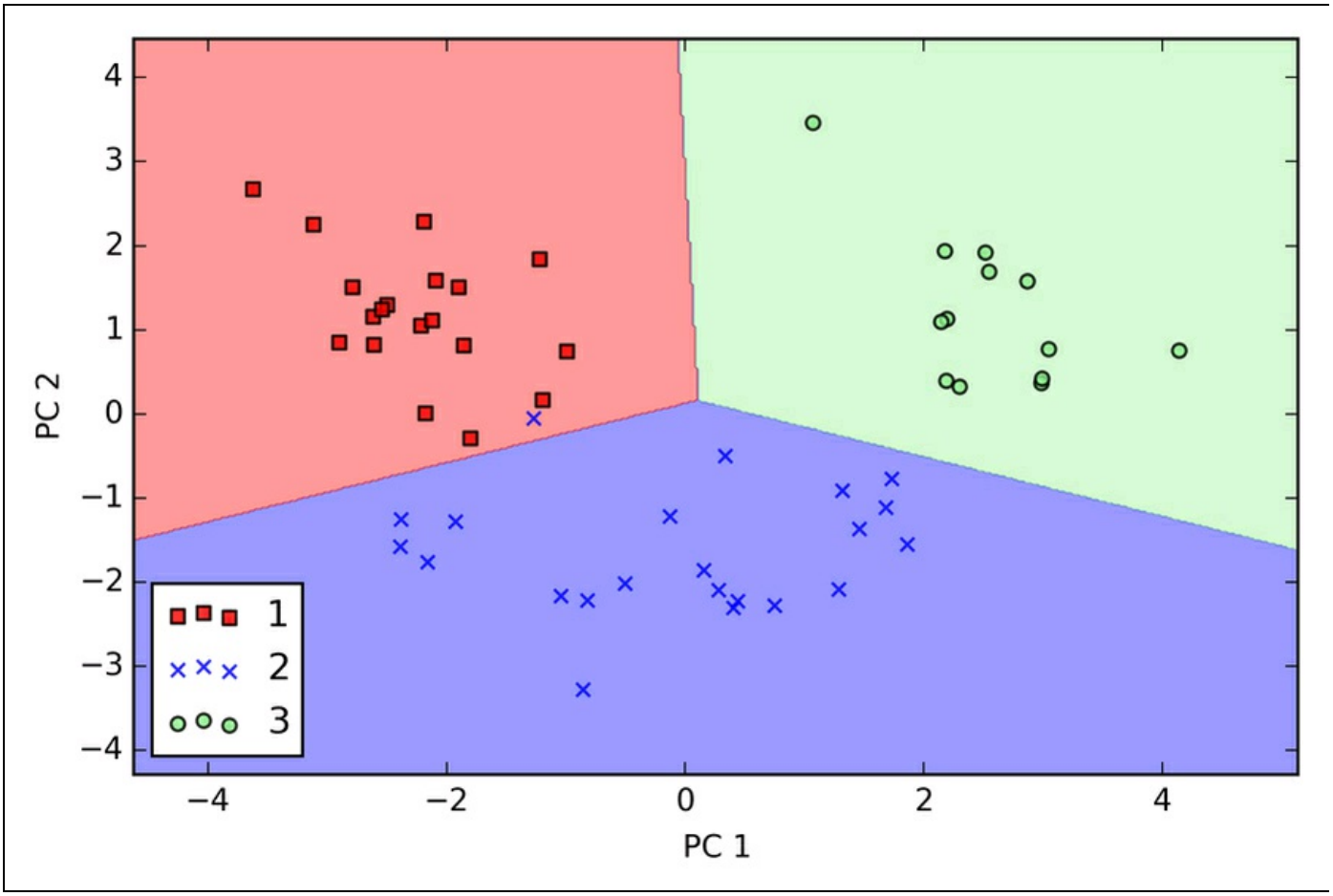


Figura 5.5

Se siamo interessati ai rapporti della varianza spiegata dei diversi componenti principali, possiamo semplicemente inizializzare la classe `PCA` con il parametro `n_components` impostato a `None`, in modo che tutti i componenti principali vengano mantenuti e il rapporto della varianza spiegata possa risultare accessibile tramite l'attributo `explained_variance_ratio_`:

```
>>> pca = PCA(n_components=None)
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> pca.explained_variance_ratio_
array([ 0.37329648,  0.18818926,  0.10896791,  0.07724389,  0.06478595,
0.04592014,  0.03986936,  0.02521914,  0.02258181,  0.01830924,
0.01635336,  0.01284271,  0.00642076])
```

Notate che abbiamo eseguito l'impostazione `n_components=None` quando abbiamo inizializzare la classe PCA, in modo che restituisse tutti i componenti principali ben ordinati, invece di eseguire la riduzione della dimensionalità.

Compressione dei dati con supervisione, tramite l'analisi discriminante lineare (LDA)

L'analisi *LDA* (*Linear Discriminant Analysis*) può essere utilizzata come tecnica per l'estrazione delle caratteristiche con lo scopo di incrementare l'efficienza computazionale e ridurre il grado di overfitting introdotto dalla maledizione della dimensionalità nei modelli non regolarizzati.

Il concetto generale su cui si basa l'analisi LDA è molto simile a quello dell'analisi PCA, tranne per il fatto che PCA cerca di trovare gli assi ortogonali dei componenti con massima varianza nel dataset. L'obiettivo dell'analisi LDA è invece quello di trovare il sottospazio delle caratteristiche che ottimizza la separabilità delle classi. Entrambe le analisi sono tecniche di trasformazione lineare, che possono essere utilizzate per ridurre il numero di dimensioni in un dataset. LDA è un algoritmo senza supervisione, mentre PCA prevede la supervisione. Pertanto, potremmo intuitivamente pensare che l'analisi LDA sia una tecnica superiore di estrazione delle caratteristiche per i compiti di classificazione, rispetto a PCA. Tuttavia, A.M. Martinez ha rilevato che la rielaborazione con PCA tende a fornire risultati di classificazione migliori in un compito di riconoscimento delle immagini in alcuni specifici casi, per esempio quando ogni classe è costituita solo da un limitato numero di campioni (A. M. Martinez e A. C. Kak. *PCA Versus LDA*. "Pattern Analysis and Machine Intelligence, IEEE Transactions", 23(2):228–233, 2001).

NOTA

Sebbene l'analisi LDA venga anche chiamata LDA di Fisher, Ronald A.Fisher ha inizialmente formulato il *discriminante lineare di Fisher* per problemi di classificazione a due classi nel 1936 (R. A. Fisher. *The Use of Multiple Measurements in Taxonomic Problems*. "Annals of Eugenics", 7(2):179–188, 1936). Il discriminante lineare di Fisher è stato generalizzato per problemi multiclasse da C. Radhakrishna Rao sotto l'assunto di covarianze uguali nelle classi e di classi distribuite in modo normale nel 1948; è questo che noi oggi chiamiamo analisi LDA (C. R. Rao. *The Utilization of Multiple Measurements in Problems of Biological Classification*. "Journal of the Royal Statistical Society. Series B (Methodological)", 10(2):159–203, 1948).

La Figura 5.6 riepiloga i concetti alla base dell'analisi LDA per un problema a due classi. I campioni della classe 1 sono rappresentati da croci e i campioni della classe 2 sono rappresentate da cerchi.

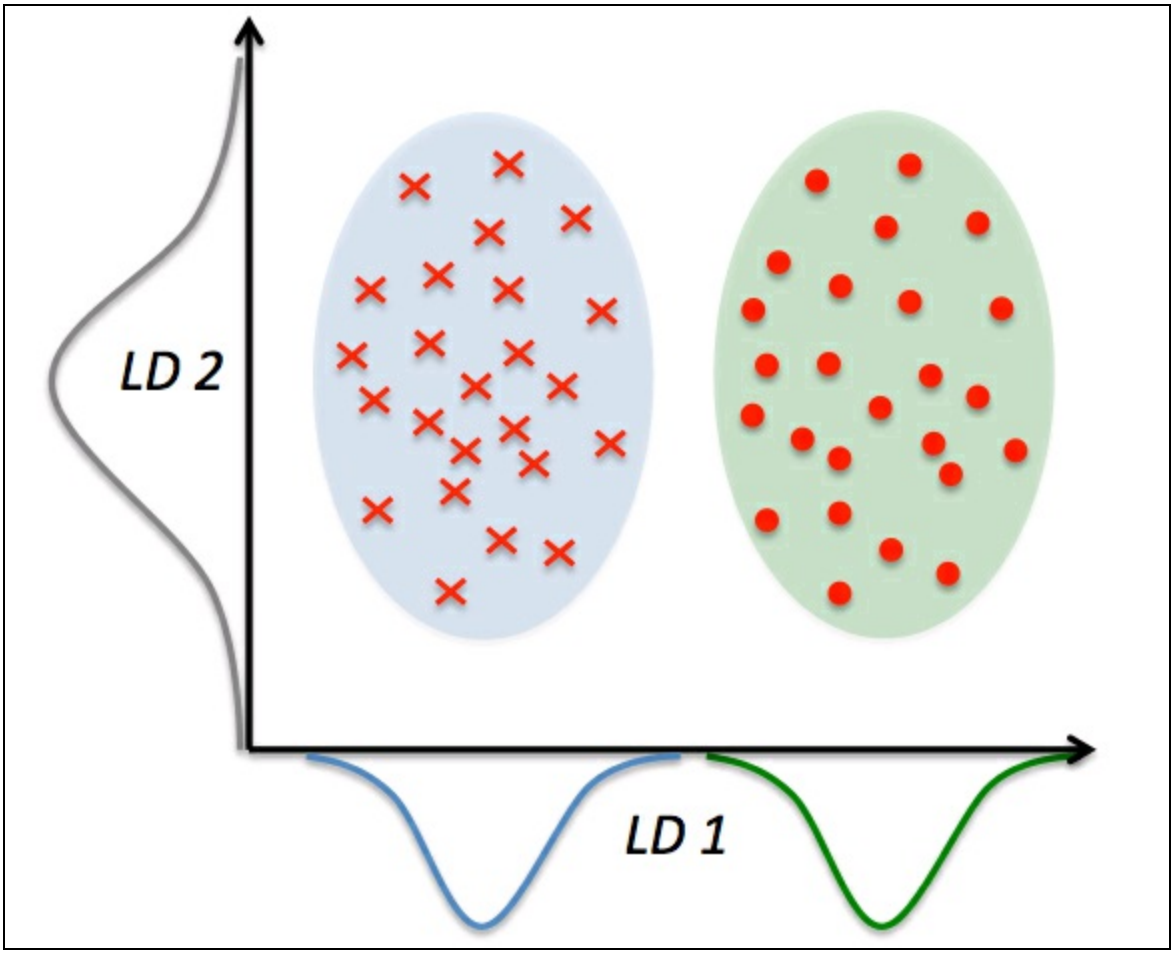


Figura 5.6

Un discriminante lineare, come illustrato nell'asse *x* (LD 1), separerebbe bene le due classi con distribuzione normale. Sebbene il discriminante lineare d'esempio rappresentato sull'asse *y* (LD 2) catturi molta della varianza del dataset, fallisce come buon discriminante lineare, in quanto non cattura nessuna delle informazioni che discriminano la classe.

Un postulato di base dell'analisi LDA è il fatto che i dati siano distribuiti in modo normale. Inoltre, si suppone che le classi abbiano matrici di covarianza identiche e che le caratteristiche siano statisticamente indipendenti l'una dall'altra. Tuttavia, anche qualora una o più di queste supposizioni venissero violate (ma solo lievemente), l'analisi LDA per la riduzione della dimensionalità sarebbe comunque in grado di funzionare piuttosto bene (R. O. Duda, P. E. Hart e D. G. Stork. *Pattern Classification*, 2ªEdizione, New York 2001).

Prima di dare un'occhiata al funzionamento interno dell'analisi LDA nei prossimi paragrafi, ricapitoliamo i passi chiave dell'approccio LDA.

1. Standardizzare il dataset d-dimensionale (d è il numero di caratteristiche).
2. Per ogni classe, calcolare il vettore d-dimensionale delle medie.

3. Costruire la matrice a dispersione fra le classi $\boldsymbol{S}_B$ e la matrice a dispersione all'interno della classe $\boldsymbol{S}_w$.
4. Calcolare gli autovettori e i corrispondenti autovalori della matrice $\boldsymbol{S}_w^{-1}\boldsymbol{S}_B$.
5. Scegliere i k autovettori che corrispondono ai k più grandi autovalori, per costruire una matrice di trasformazione $d \times k$-dimensionale $\boldsymbol{W}$; gli autovettori sono le colonne della matrice.
6. Proiettare i campioni sul nuovo sottospazio delle caratteristiche, utilizzando la matrice di trasformazione $\boldsymbol{W}$.

NOTA

Le supposizioni che applichiamo quando utilizziamo l'analisi LDA sono che le caratteristiche siano distribuite in modo normale e indipendenti fra loro. Inoltre, l'algoritmo LDA presuppone che le matrici di covarianza delle singole classi siano identiche. Tuttavia, come abbiamo detto, anche se violiamo leggermente questi presupposti, l'analisi LDA si comporta ragionevolmente bene in entrambi i compiti: la riduzione della dimensionalità e i compiti di classificazione e la classificazione (R. O. Duda, P. E. Hart e D. G. Stork. *Pattern Classification*, 2ª. Edizione, New York 2001).

Calcolo delle matrici a dispersione

Poiché abbiamo già standardizzato le caratteristiche del dataset *Wine* nella sezione dedicata all'analisi PCA all'inizio di questo capitolo, possiamo saltare il primo passo e procedere con il calcolo dei vettori delle medie, che utilizzeremo per costruire le matrici a dispersione delle classi e fra le classi. Ogni vettore delle medie $\boldsymbol{m}_i$ conserva il valore medio della caratteristica μ_m, rispetto ai campioni della classe i:

$$\boldsymbol{m}_i = \frac{1}{n_i} \sum_{\boldsymbol{x} \in D_i}^{c} \boldsymbol{x}_m$$

Questo produce tre vettori delle medie:

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix} \quad i \in \{1,2,3\}$$

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(
...                X_train_std[y_train==label], axis=0))
...     print('MV %s: %s\n' %(label, mean_vecs[label-1]))
MV 1: [ 0.9259 -0.3091  0.2592 -0.7989  0.3039  0.9608  1.0515 -0.6306  0.5354
  0.2209  0.4855  0.798   1.2017]
MV 2: [-0.8727 -0.3854 -0.4437  0.2481 -0.2409 -0.1059  0.0187 -0.0164  0.1095
 -0.8796  0.4392  0.2776 -0.7016]
MV 3: [ 0.1637  0.8929  0.3249  0.5658 -0.01   -0.9499 -1.228   0.7436 -0.7652
  0.979  -1.1698 -1.3007 -0.3912]
```

Utilizzando i vettori delle medie, possiamo ora calcolare la matrice a dispersione all'interno della classe S_W:

$$S_W = \sum_{i=1}^{c} S_i$$

Questa viene calcolata sommando le singole matrici a dispersione S_i di ogni singola classe i:

$$S_i = \sum_{x \in D_i}^{c} (x - m_i)(x - m_i)^T$$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1,4), mean_vecs):
...     class_scatter = np.zeros((d, d))
...     for row in X_train[y_train == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row-mv).dot((row-mv).T)
...     S_W += class_scatter
>>> print('Within-class scatter matrix: %sx%s'
...       % (S_W.shape[0], S_W.shape[1]))
Within-class scatter matrix: 13x13
```

La supposizione che stiamo applicando quando calcoliamo le matrici a dispersione è il fatto che le etichette delle classi nel set di addestramento siano distribuite in modo uniforme. Tuttavia, se stampiamo il numero delle etichette delle classi, scopriamo che questa supposizione viene violata:

```
>>> print('Class label distribution: %s'
...       % np.bincount(y_train)[1:])
Class label distribution: [40 49 35]
```

Pertanto, vogliamo cambiare la scala delle singole matrici a dispersione S_i prima di sommarle nella matrice a dispersione S_w. Quando dividiamo le matrici a dispersione per il numero dei campioni della classe N_i, possiamo vedere che calcolare la matrice a dispersione è in realtà la stessa cosa che calcolare la matrice di covarianza Σ_i. La matrice di covarianza è una versione normalizzata della matrice a dispersione:

$$\Sigma_i = \frac{1}{N_i} \boldsymbol{S}_W = \frac{1}{N_i} \sum_{\boldsymbol{x} \in D_i}^{c} \left(\boldsymbol{x} - \boldsymbol{m}_i\right)\left(\boldsymbol{x} - \boldsymbol{m}_i\right)^T$$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.cov(X_train_std[y_train==label].T)
...     S_W += class_scatter
>>> print('Scaled within-class scatter matrix: %sx%s'
...       % (S_W.shape[0], S_W.shape[1]))
Scaled within-class scatter matrix: 13x13
```

Dopo aver calcolato la matrice a dispersione della classe in scala (ovvero la matrice di covarianza), possiamo procedere al passo successivo e calcolare la matrice a dispersione fra le classi $\boldsymbol{S}_B$:

$$\boldsymbol{S}_B = \sum_{i=1}^{c} N_i \left(\boldsymbol{m}_i - \boldsymbol{m}\right)\left(\boldsymbol{m}_i - \boldsymbol{m}\right)^T$$

Qui, $\boldsymbol{m}$ è la media generale, che è stata calcolata comprendendo i campioni di tutte le classi.

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> d = 13 # number of features
>>> S_B = np.zeros((d, d))
>>> for i,mean_vec in enumerate(mean_vecs):
...     n = X_train[y_train==i+1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1)
...     mean_overall = mean_overall.reshape(d, 1)
    S_B += n * (mean_vec - mean_overall).dot(
...               (mean_vec - mean_overall).T)
print('Between-class scatter matrix: %sx%s'
...    % (S_B.shape[0], S_B.shape[1]))
Between-class scatter matrix: 13x13
```

Selezione dei discriminanti lineari per il nuovo sottospazio delle caratteristiche

I passi rimanenti dell'analisi LDA sono simili a quelli dell'analisi PCA. Tuttavia, invece di calcolare l'autodecomposizione sulla matrice di covarianza, risolviamo il problema dell'autovalore generalizzato della matrice $\boldsymbol{S}_w^{-1}\boldsymbol{S}_B$:

```
>>> eigen_vals, eigen_vecs =\
...np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

Dopo aver calcolato le autocoppie, possiamo ordinare gli autovalori in senso discendente:

```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...               for i in range(len(eigen_vals))]
>>> eigen_pairs = sorted(eigen_pairs,
...               key=lambda k: k[0], reverse=True)
>>> print('Eigenvalues in decreasing order:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])
```

```
Eigenvalues in decreasing order:
452.721581245
156.43636122
8.11327596465e-14
2.78687384543e-14
2.78687384543e-14
2.27622032758e-14
2.27622032758e-14
1.97162599817e-14
1.32484714652e-14
1.32484714652e-14
1.03791501611e-14
5.94140664834e-15
2.12636975748e-16
```

Nell'analisi LDA, il numero di discriminanti lineari è al più $c-1$, dove C è il numero delle etichette delle classi, in quanto la matrice a dispersione delle classi S_B è la somma di C matrici con rango 1 o inferiore. Possiamo pertanto vedere che abbiamo solo due autovalori diversi da 0 (gli autovalori 3-13 non sono esattamente 0, ma ciò è dovuto solo all'aritmetica in virgola mobile di NumPy). Notate che nel raro caso di colinearità perfetta (tutti i punti del campione cadono su una linea retta), la matrice di covarianza avrà rango 1, il che produrrà un solo autovettore con un autovalore diverso da 0.

Per misurare quanta parte dell'informazione discriminatoria della classe venga catturata dai discriminanti lineari (autovettori), tracciamo i discriminanti lineari riducendo gli autovalori in modo simile a quanto già visto nel grafico della varianza spiegata che abbiamo creato nella sezione dedicata all'analisi PCA. Per semplicità, chiameremo il contenuto dell'informazione discriminativa della classe con il termine di *discriminabilità*.

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real, reverse=True)]
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, alpha=0.5, align='center',
...         label='individual "discriminability"')
>>> plt.step(range(1, 14), cum_discr, where='mid',
...          label='cumulative "discriminability"')
>>> plt.ylabel('"discriminability" ratio')
>>> plt.xlabel('Linear Discriminants')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.show()
```

Come possiamo vedere dalla Figura 5.7, i primi due discriminanti lineari catturano praticamente il 100% delle informazioni utili del dataset di addestramento *Wine*.

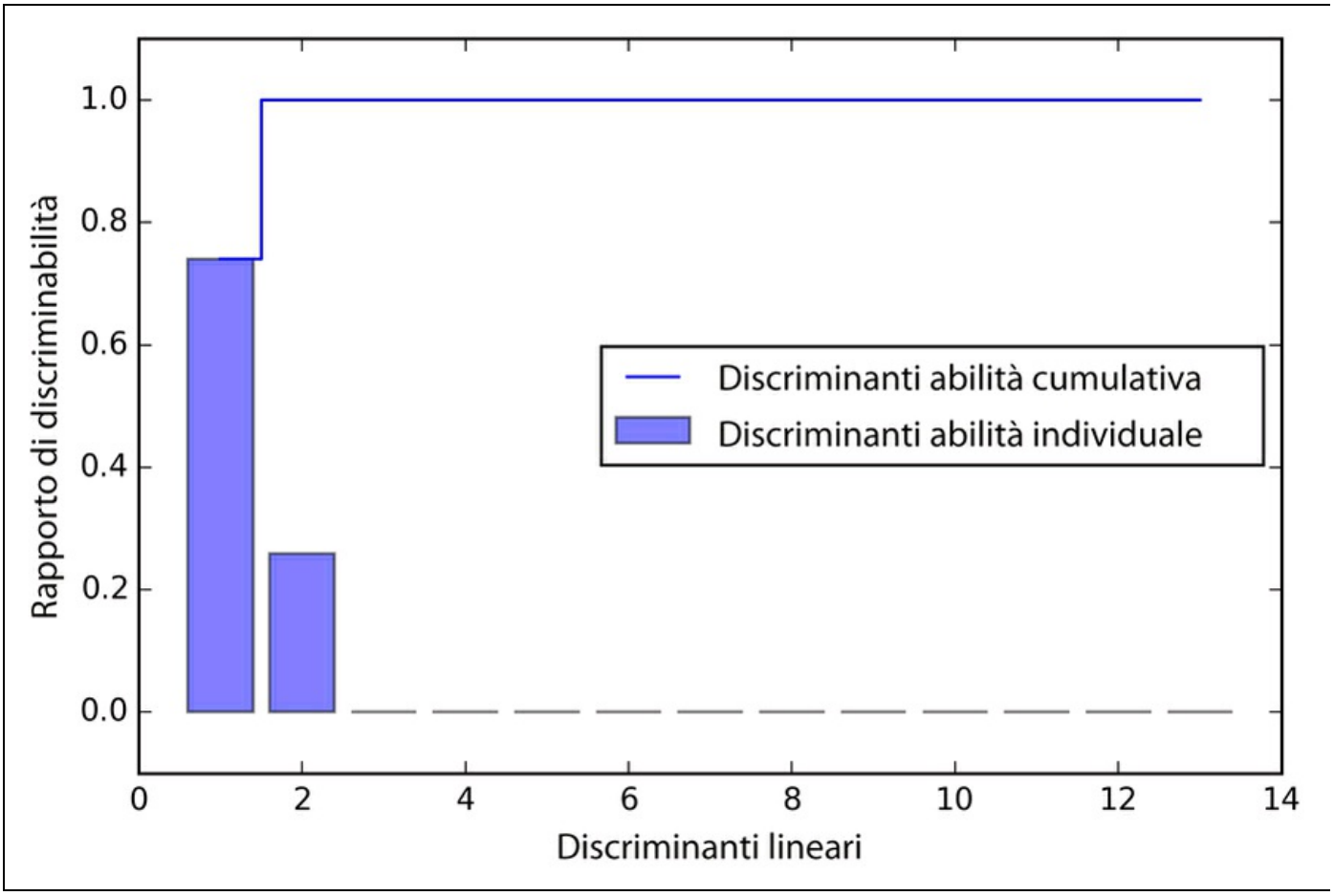


Figura 5.7

Impiliamo le due colonne di autovettori maggiormente discriminativi per creare la matrice di trasformazione W:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w)
Matrix W:
[[ 0.0662 -0.3797]
[-0.0386 -0.2206]
[ 0.0217 -0.3816]
[-0.184 0.3018]
[ 0.0034 0.0141]
[-0.2326 0.0234]
[ 0.7747 0.1869]
[ 0.0811 0.0696]
[-0.0875 0.1796]
[-0.185 -0.284 ]
[ 0.066 0.2349]
[ 0.3805 0.073 ]
[ 0.3285 -0.5971]]
```

Proiezione dei campioni sul nuovo spazio di caratteristiche

Utilizzando la matrice di trasformazione W che abbiamo creato nel paragrafo precedente, possiamo ora trasformare i dati di addestramento moltiplicando le

matrici:

$$X' = XW$$

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0]*(-1)
...                 X_train_lda[y_train==l, 1]*(-1)
...                 c=c, label=l, marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

Come possiamo vedere nel grafico della Figura 5.8, le tre classi di vini sono ora separabili linearmente nel nuovo sottospazio delle caratteristiche.

Analisi LDA con scikit-learn

L'implementazione a passi è stata un buon esercizio per comprendere il funzionamento dell'analisi LDA e per comprendere le differenze esistenti fra le due analisi LDA e PCA.

Ora diamo un'occhiata alla classe LDA implementata in scikit-learn:

```
>>> from sklearn.lda import LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

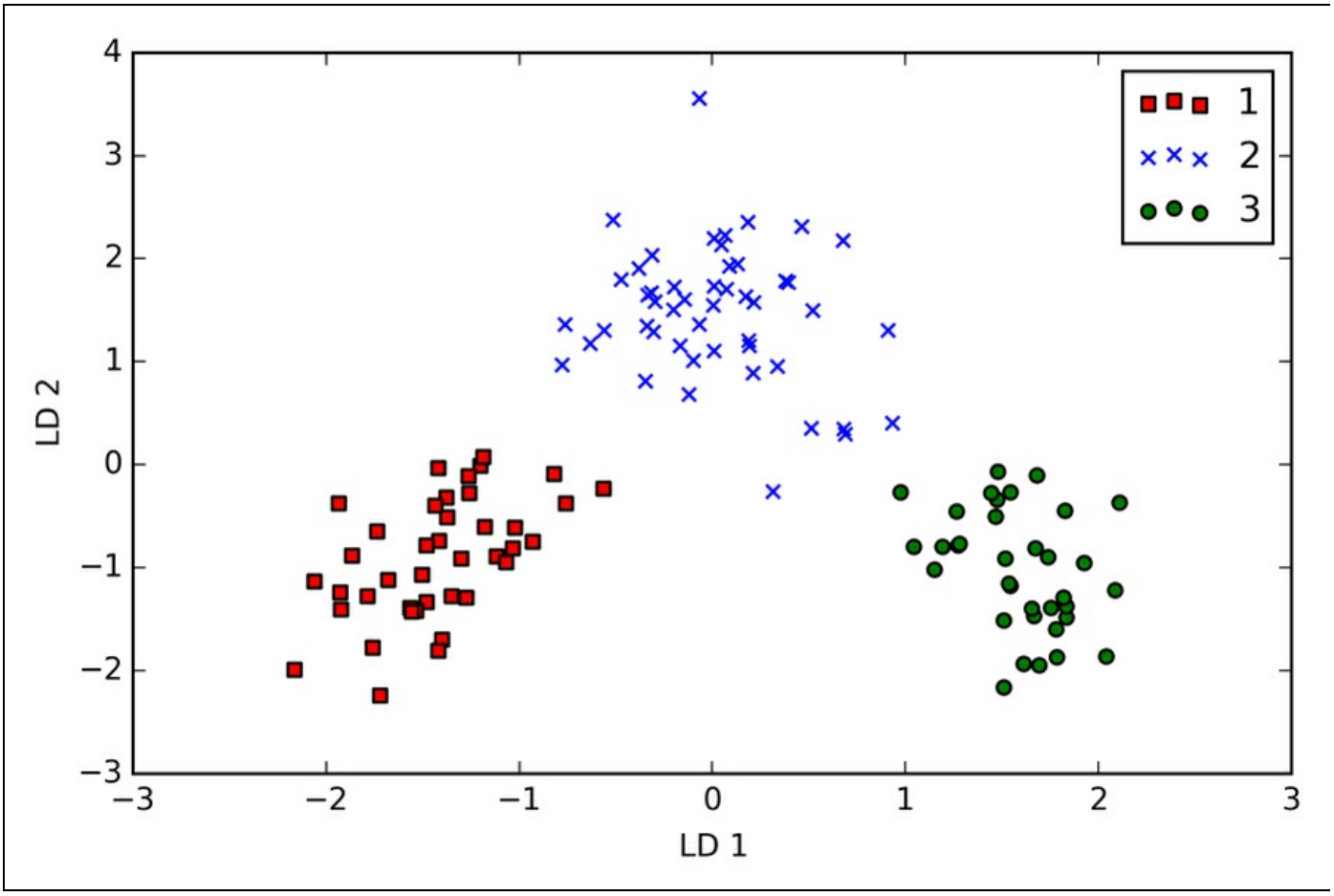


Figura 5.8

Poi vediamo come il classificatore a regressione logistica gestisce il dataset di addestramento di dimensioni inferiori, dopo la trasformazione LDA:

```
>>> lr = LogisticRegression()
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Osservando il grafico risultante (Figura 5.9), vediamo che il modello a regressione logistica sbaglia a classificare uno dei campioni della classe 2.

Riducendo l'intensità della regolarizzazione, potremmo probabilmente spostare i confini decisionali in modo che i modelli a regressione logistica siano in grado di classificare tutti i campioni del dataset di addestramento in modo corretto. Tuttavia, diamo un'occhiata ai risultati del set di test:

```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Come vediamo nella Figura 5.10, il classificatore a regressione logistica è in grado di ottenere un punteggio di accuratezza perfetto per la classificazione dei

campioni del dataset di test, utilizzando solo un sottospazio bidimensionale delle caratteristiche, invece delle tredici caratteristiche originali del dataset *Wine*.

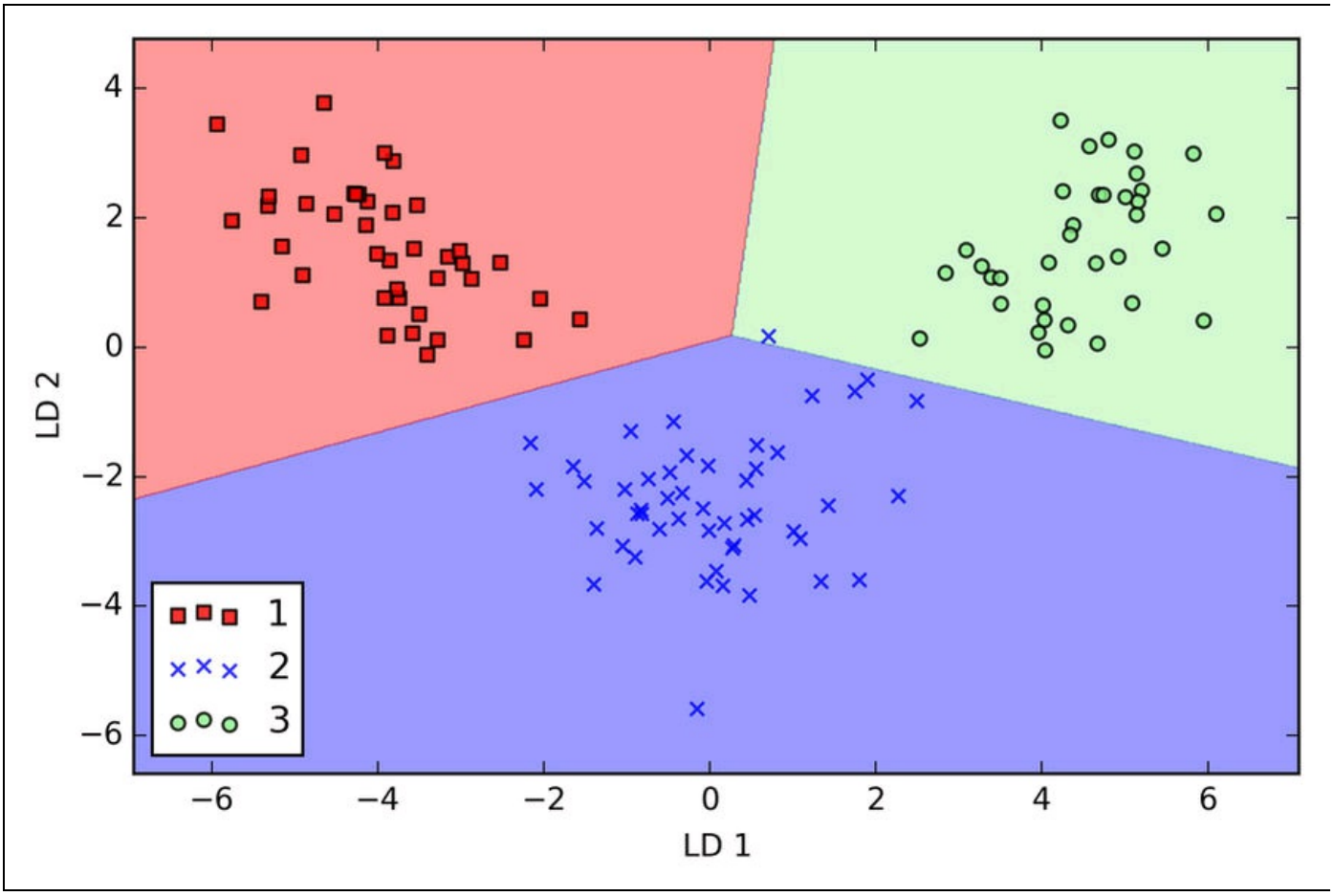


Figura 5.9

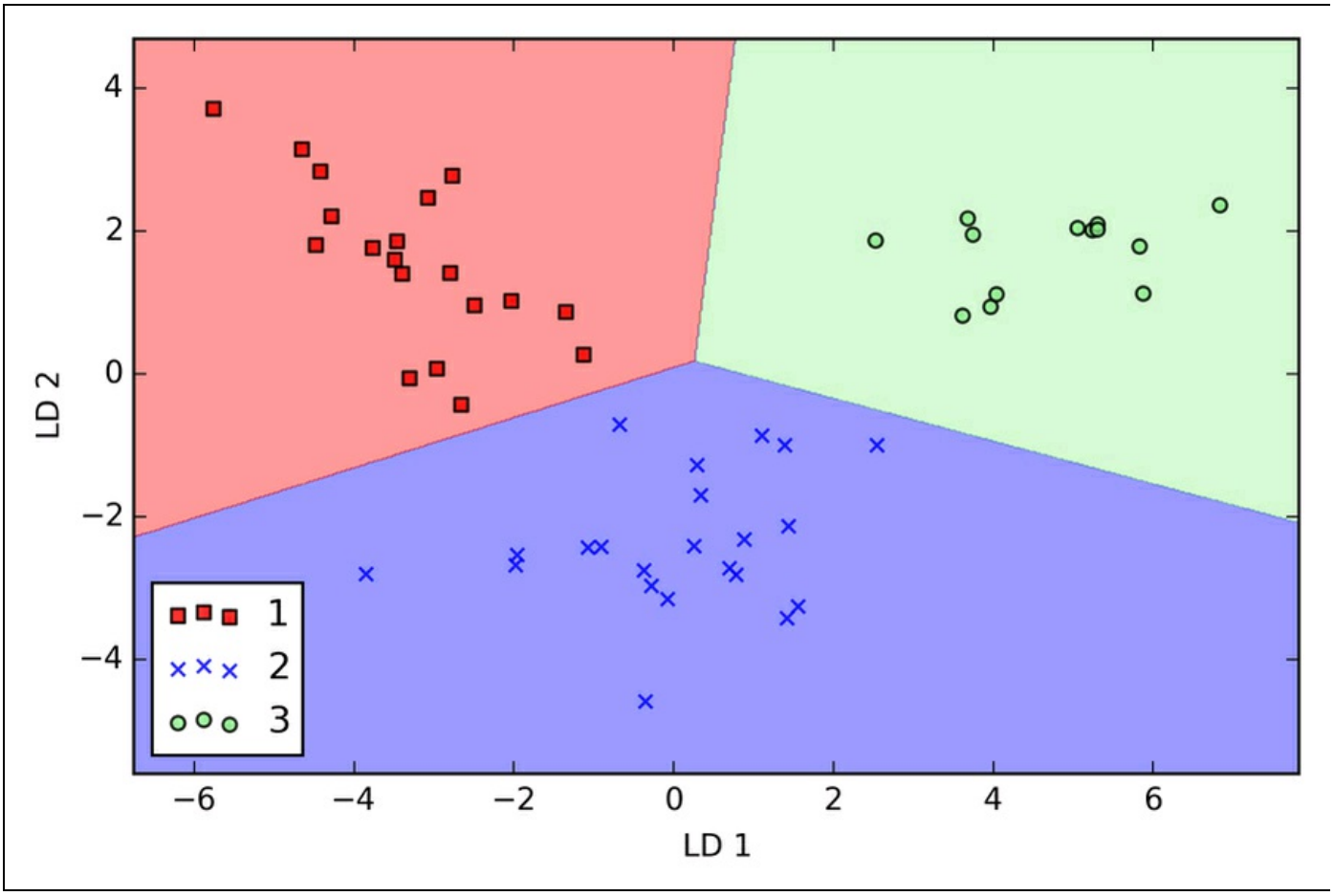


Figura 5.10

Uso della kernel PCA per il mappaggio non lineare

Molti algoritmi di machine learning si basano sul presupposto della separabilità lineare dei dati. Come abbiamo imparato, il perceptron richiede addirittura obbligatoriamente che i dati di addestramento siano separabili in modo lineare per poter convergere.

Altri algoritmi che abbiamo trattato finora presuppongono che la mancanza di una separabilità lineare perfetta sia dovuta al rumore: Adaline, regressione logistica e SVM (*Support Vector Machine*) in versione standard solo per nominarne alcuni. Tuttavia, se dobbiamo occuparci di problemi non lineari, che possiamo incontrare con una certa frequenza delle applicazioni del mondo reale, le tecniche di trasformazione lineare per la riduzione della dimensionalità, come PCA e LDA, possono non essere la scelta migliore. In questo paragrafo osserveremo una versione a kernel dell'analisi PCA o *kernel PCA*, che fa riferimento ai concetti già trattati per le SVM e che abbiamo esaminato nel Capitolo 3, *I classificatori di machine learning di scikit-learn*. Utilizzando, l'analisi kernel PCA, possiamo trasformare dei dati che non siano separabili in modo lineare trasportandoli in un nuovo sottospazio di dimensioni inferiori che risulti adatto per i classificatori lineari (Figura 5.11).

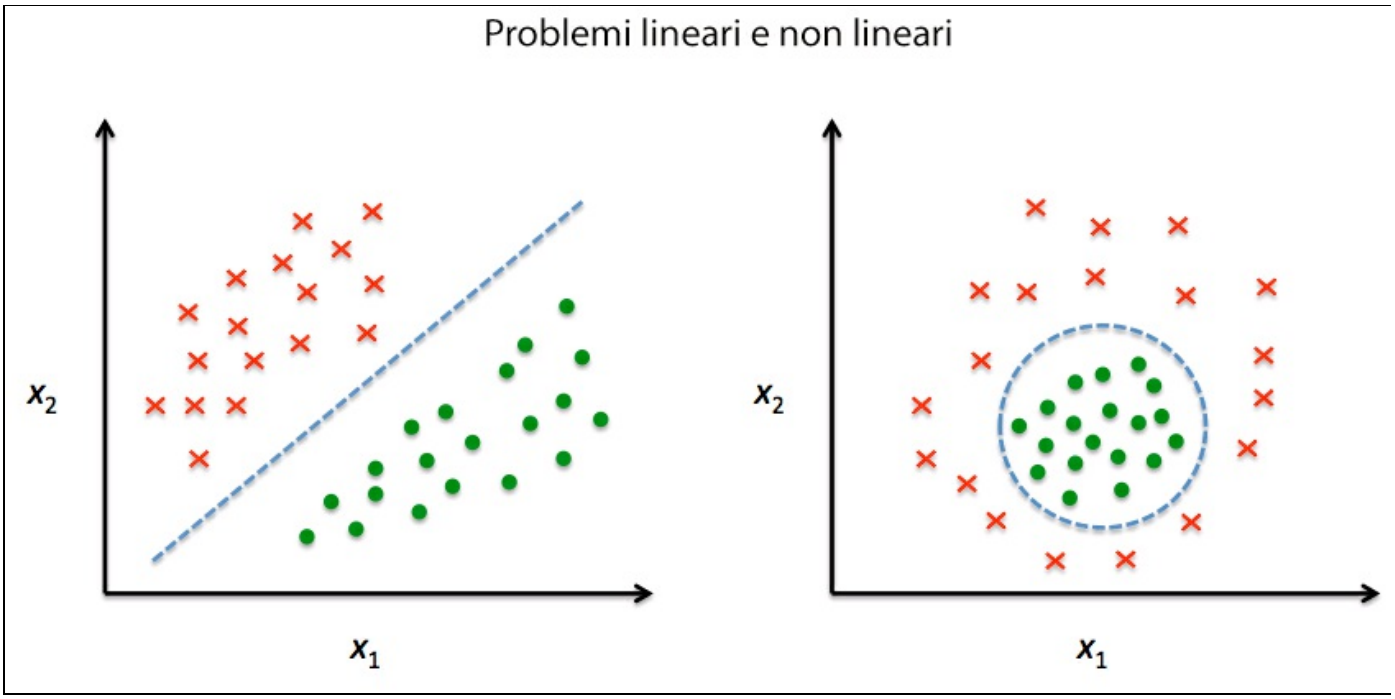


Figura 5.11

Funzioni kernel e tecniche kernel

Come possiamo ricordare dalla discussione relativa alle SVM kernel del Capitolo 3, *I classificatori di machine learning di scikit-learn*, possiamo affrontare dei problemi non lineari proiettandoli su un nuovo spazio di caratteristiche di dimensionalità superiore, dove le classi divengano separabili in modo lineare. Per trasformare i campioni $\boldsymbol{x} \in \mathbb{R}^d$ in questo sottospazio k-dimensionale di maggiori dimensioni, abbiamo definito una funzione di mappaggio non lineare ϕ:

$$\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k \quad (k >> d)$$

Possiamo considerare ϕ come una funzione che crea combinazioni non lineari delle caratteristiche originali, con lo scopo di mappare il dataset d-dimensionale originale in uno spazio di caratteristiche di maggiori dimensioni, k-dimensionale. Per esempio, se avessimo un vettore di caratteristiche $\boldsymbol{x} \in \mathbb{R}^d$ ($\boldsymbol{x}$ è un vettore colonna costituito da d caratteristiche) con due dimensioni (d = 2), un potenziale mappaggio in uno spazio tridimensionale potrebbe essere il seguente:

$$\boldsymbol{x} = \left[x_1,\ x_2 \right]^T$$

$$\downarrow \phi$$

$$\mathbf{z} = \left[x_1^2, \sqrt{2 x_1 x_2}, x_2^2 \right]^T$$

In altre parole, tramite l'analisi PCA *kernel* eseguiamo un mappaggio non lineare che trasforma i dati in uno spazio a maggiori dimensioni e utilizziamo poi l'analisi PCA *standard* in questo spazio di maggiori dimensioni per proiettare i dati di nuovo su uno spazio a minori dimensioni, dove i campioni possano essere separati tramite un classificatore lineare (sotto la condizione che i campioni possano essere separati per densità nello spazio di input). Tuttavia, un difetto di questo approccio è il fatto che è molto costoso dal punto di vista computazionale e per questo utilizziamo la *tecnica kernel*. Utilizzando la tecnica kernel, possiamo calcolare la similarità fra due vettori di caratteristiche a elevate dimensioni nello spazio delle caratteristiche originario.

Prima di procedere con i dettagli sull'uso della tecnica kernel per affrontare questo problema costoso dal punto di vista computazionale, torniamo all'approccio

PCA *standard* che abbiamo implementato all'inizio di questo capitolo. Abbiamo calcolato la covarianza fra due caratteristiche k e j nel seguente modo:

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

Poiché la standardizzazione delle caratteristiche le centra alla media zero, per esempio

$$\frac{1}{n}\sum_i x_j^{(i)} = 0$$

possiamo semplificare questa equazione nel seguente modo:

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n} x_j^{(i)} x_k^{(i)}$$

Notate che l'equazione precedente fa riferimento alla covarianza fra le due caratteristiche; ora, scriviamo l'equazione generale per calcolare la *matrice* di covarianza Σ:

$$\sum = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}^{(i)}\, \boldsymbol{x}^{(i)T}$$

Bernhard Scholkopf ha generalizzato questo approccio (B. Scholkopf, A. Smola e K. R. Muller, *Kernel Principal Component Analysis*, pp. 583–588, 1997) in modo che potessimo sostituire i prodotti fra i campioni nello spazio originario delle caratteristiche con le combinazioni delle caratteristiche non lineari tramite ϕ:

$$\sum = \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi(\boldsymbol{x}^{(i)})^T$$

Per ottenere gli autovettori (i componenti principali) da questa matrice di covarianza, dobbiamo risolvere la seguente equazione:

$$\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow \frac{1}{n}\sum_{i=1}^{n}\phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T}\boldsymbol{v}=\lambda\boldsymbol{v}$$

$$\Rightarrow v=\frac{1}{n\lambda}\sum_{i=1}^{n}\phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T}\boldsymbol{v}=\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{a}^{(i)}\phi\left(\boldsymbol{x}^{(i)}\right)$$

Qui, λ e $\boldsymbol{v}$sono gli autovalori e gli autovettori della matrice di covarianza Σ e $\boldsymbol{a}$ può essere ottenuto estraendo gli autovettori della matrice kernel (di similarità) $\boldsymbol{K}$, come vedremo nei prossimi paragrafi.

La derivata della matrice kernel è descritta di seguito.

Innanzitutto, scriviamo la matrice di covarianza in notazione a matrice, dove $\phi(X)$ è una matrice $n\times k$-dimensionale:

$$\sum=\frac{1}{n}\sum_{i=1}^{n}\phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T}=\frac{1}{n}\phi(\boldsymbol{X})^{T}\phi(\boldsymbol{X})$$

Ora, possiamo scrivere l'equazione ad autovettore nel seguente modo:

$$v=\frac{1}{n}\sum_{i=1}^{n}a^{(i)}\phi\left(\boldsymbol{x}^{(i)}\right)=\lambda\phi(\boldsymbol{X})^{T}\boldsymbol{a}$$

Poiché $\Sigma\boldsymbol{v}=\lambda\boldsymbol{v}$, otteniamo:

$$\frac{1}{n}\phi(\boldsymbol{X})^{T}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\boldsymbol{a}=\lambda\phi(\boldsymbol{X})^{T}\boldsymbol{a}$$

Moltiplicandola per $\phi(X)$ su entrambi i lati si ottiene il seguente risultato:

$$\frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\boldsymbol{a}=\lambda\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\boldsymbol{a}=\lambda\boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\boldsymbol{K}\boldsymbol{a} = \lambda \boldsymbol{a}$$

Qui, $\boldsymbol{K}$ è la matrice di similarità (kernel):

$$\boldsymbol{K} = \phi(\boldsymbol{X})\phi(\boldsymbol{X})^T$$

Come ricorderemo dalla sezione dedicata alle SVM nel Capitolo 3, *I classificatori di machine learning di scikit-learn*, utilizziamo la tecnica del kernel per evitare di calcolare esplicitamente i prodotti a coppie dei campioni $\boldsymbol{x}$ sotto ϕ. Utilizziamo una funzione kernel $\boldsymbol{K}$ in modo che non sia necessario calcolare esplicitamente gli autovettori:

$$k\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \phi\left(\boldsymbol{x}^{(i)}\right)^T \phi\left(\boldsymbol{x}^{(j)}\right)$$

In altre parole, ciò che otteniamo dopo l'analisi kernel PCA sono i campioni già proiettati sui rispettivi componenti invece di costruire una matrice di trasformazione come accade nell'approccio PCA standard. Fondamentalmente, la funzione kernel può essere considerata come una funzione che calcola un prodotto fra due vettori, una misura di similarità.

I kernel più comunemente utilizzati sono i seguenti.

- Kernel polinomiale:

$$k\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \left(\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)} + \theta\right)^p$$

Qui, θ è la soglia e p è la potenza, che deve essere specificata dall'utente.

- Kernel a tangente iperbolica (sigmoid):

$$k\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \tanh\left(\eta \boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)} + \theta\right)$$

- Kernel *Radial Basis Function* (*RBF*) o gaussiana che utilizzeremo nei prossimi esempi, presentati nel prossimo paragrafo:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^2}{2\sigma^2}\right)$$

che può essere scritta anche nel seguente modo:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\gamma\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^2\right)$$

Per riepilogare il tutto, possiamo definire i seguenti tre passi per implementare un'analisi kernel PCA RBF.

1. Calcoliamo la matrice kernel (di similarità) k, dove dobbiamo eseguire il seguente calcolo:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\gamma\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^2\right)$$

   Lo facciamo per ogni coppia di campioni:

$$\boldsymbol{K}=\begin{bmatrix}\kappa\left(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(1)}\right) & \kappa\left(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(n)}\right)\\ \kappa\left(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(1)}\right) & \left(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(n)}\right)\\ \vdots & \vdots & \ddots & \vdots\\ \kappa\left(\boldsymbol{x}^{(n)},\boldsymbol{x}^{(1)}\right) & \kappa\left(\boldsymbol{x}^{(n)},\boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(n)},\boldsymbol{x}^{(n)}\right)\end{bmatrix}$$

   Per esempio, se il nostro dataset contiene 100 campioni di addestramento, la matrice kernel simmetrica delle similarità a coppie sarebbe di dimensioni pari a 100×100.

2. Centriamo la matrice kernel k utilizzando la seguente equazione:

$$\boldsymbol{K}'=\boldsymbol{K}-1_n\boldsymbol{K}-\boldsymbol{K}1_n+1_n\boldsymbol{K}1_n$$

Qui, 1_n è una matrice $n \times n$-dimensionale (le stesse dimensioni della matrice kernel) dove tutti i valori sono uguali a $\frac{1}{n}$.

3. Raccogliamo i k autovettori superiori della matrice kernel centrata sulla base dei corrispondenti autovalori, che vengono valutati per dimensioni decrescenti. Rispetto all'analisi PCA standard, gli autovettori non sono gli assi dei componenti principali, ma i campioni proiettati su quegli assi.

A questo punto, potreste chiedervi perché dobbiamo centrare la matrice kernel nel secondo passo. Precedentemente siamo partiti dalla supposizione che stessimo lavorando su dati standardizzati, dove tutte le caratteristiche hanno media 0 quando abbiamo formulato la matrice di covarianza e abbiamo sostituito i prodotti per le combinazioni non lineari della caratteristica tramite ϕ. Pertanto, la centratura della matrice kernel del secondo passo diviene necessaria, in quanto non calcoliamo esplicitamente il nuovo spazio della caratteristica e non possiamo garantire che il nuovo spazio della caratteristica sia centrato sullo 0.

Nel prossimo paragrafo, metteremo in pratica questi tre passi, implementando un'analisi PCA kernel in Python.

Implementazione di una kernel PCA in Python

Nei paragrafi precedenti abbiamo trattato i concetti su cui si basa l'analisi PCA kernel. Ora dobbiamo implementare in Python un'analisi PCA kernel RBF seguendo i tre passi che abbiamo riassunto nell'approccio PCA kernel. Utilizzando le funzioni di supporto di SciPy e NumPy, vedremo che l'implementazione dell'analisi PCA kernel è in realtà molto semplice:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np
def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.
    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_samples, n_features]
    gamma: float
      Tuning parameter of the RBF kernel
    n_components: int
      Number of principal components to return
    Returns
    ------------
     X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
       Projected dataset
    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')
    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)
```

```
    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)
    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)
    # Obtaining eigenpairs from the centered kernel matrix
    # numpy.eigh returns them in sorted order
    eigvals, eigvecs = eigh(K)
    # Collect the top k eigenvectors (projected samples)
    X_pc = np.column_stack((eigvecs[:, -i]
                        for i in range(1, n_components + 1)))
    return X_pc
```

Un difetto dell'uso dell'analisi PCA kernel RBF per la riduzione della dimensionalità è il fatto che dobbiamo specificare a priori il parametro γ. La ricerca di un valore appropriato per γ richiede un po' di sperimentazione, che può essere svolta al meglio utilizzando algoritmi di ottimizzazione del parametro, per esempio la ricerca a griglia, che tratteremo in dettaglio nel Capitolo 6, *Valutazione dei modelli e ottimizzazione degli iperparametri*.

Esempio 1 – Separazione di forme a mezzaluna

Ora applichiamo `rbf_kernel_pca` su alcuni dataset d'esempio non lineari. Inizieremo creando un dataset bidimensionale di 100 punti campione che producono due forme a mezzaluna:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

Per gli scopi dell'illustrazione, la mezzaluna dei simboli triangolari rappresenterà una classe e la mezzaluna dei simboli circolari rappresenterà i campioni dell'altra classe (Figura 5.12).

Chiaramente, queste due forme a mezzaluna non sono separabili linearmente e il nostro scopo è quello di "aprire" le mezzelune tramite l'analisi PCA kernel in modo che il nuovo dataset possa fungere da input adatto per un classificatore lineare. Ma innanzitutto vediamo l'aspetto del dataset se viene proiettato sui componenti principali tramite l'analisi PCA standard:

```
>>> from sklearn.decomposition import PCA
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...             color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...             color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
```

```
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

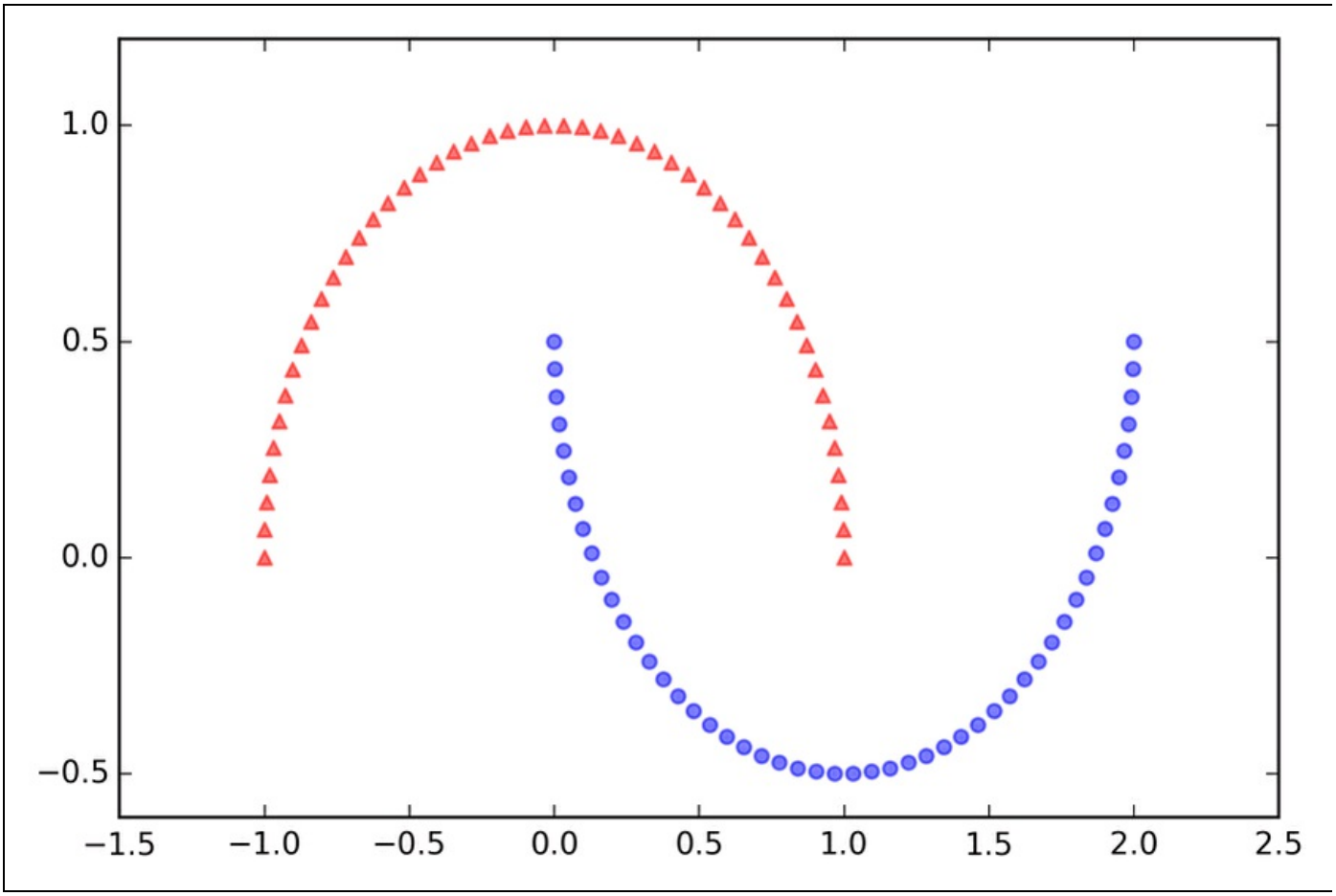


Figura 5.12

Chiaramente, possiamo vedere nella figura risultante che un classificatore lineare non sarebbe assolutamente in grado di comportarsi correttamente sul dataset trasformato tramite un'analisi PCA standard (Figura 5.13).

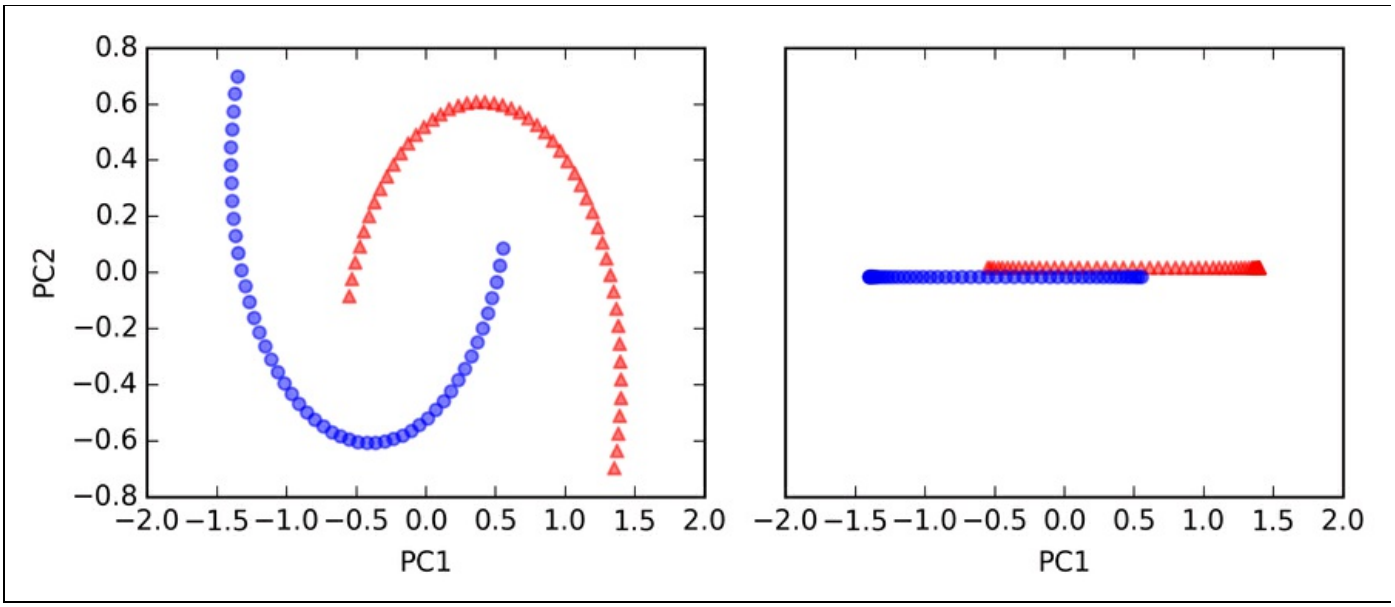


Figura 5.13

Notate che nel tracciato del solo primo componente principale (grafico a destra), abbiamo alzato leggermente i campioni triangolari e abbassato leggermente i campioni circolari, con il solo scopo di visualizzarli, altrimenti i campioni si sarebbero sovrapposti.

NOTA

Ricordate che l'analisi PCA è un metodo senza supervisione e non utilizza le informazioni sulle etichette delle classi per massimizzare la varianza come invece fa l'analisi LDA. Qui, i simboli triangolari e circolari sono stati semplicemente aggiunti per scopi di visualizzazione, in modo da indicare il grado di separazione.

Ora, tentiamo di utilizzare la funzione PCA kernel `rbf_kernel_pca`, che abbiamo implementato nel paragrafo precedente:

```
>>> from matplotlib.ticker import FormatStrFormatter
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> ax[0].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> ax[1].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> plt.show()
```

Possiamo vedere che le due classi (cerchi e triangoli) ora sono separabili linearmente e dunque il dataset di addestramento diviene adatto all'impiego di un classificatore lineare (Figura 5.14).

Sfortunatamente non esiste alcun valore universale del parametro di ottimizzazione γ che funzioni al meglio con dataset differenti. Per trovare un valore γ che sia appropriato per un determinato problema è necessario sperimentare. Nel Capitolo 6, *Valutazione dei modelli e ottimizzazione degli iperparametri*, tratteremo le tecniche che possono aiutarci ad automatizzare il compito di individuare tali parametri di ottimizzazione. Qui utilizzeremo i valori di γ che sappiamo già che forniscono *buoni* risultati.

Esempio 2 – Separazione di cerchi concentrici

Nel paragrafo precedente abbiamo visto come separare forme a mezzaluna tramite l'analisi PCA kernel. Poiché abbiamo dedicato tanto impegno a conoscere i

concetti dell'analisi PCA kernel, diamo un'occhiata a un altro interessante problema non lineare: i cerchi concentrici.

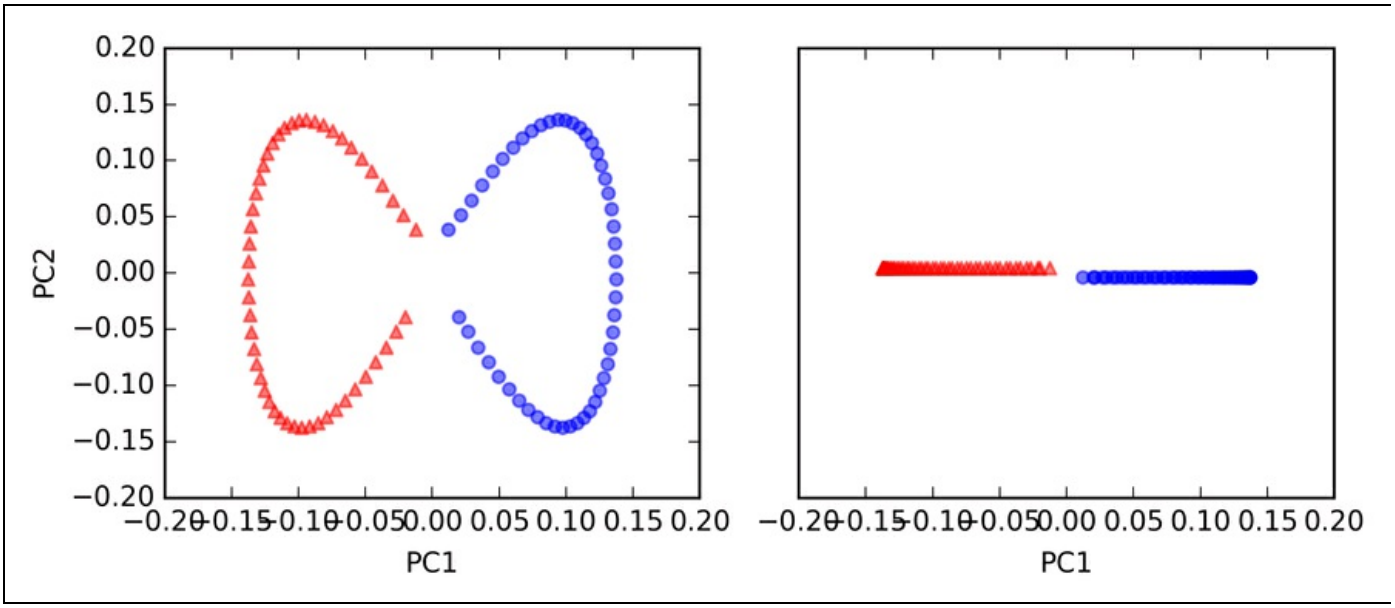


Figura 5.14

Il codice è il seguente:

```
>>> from sklearn.datasets import make_circles
>>> X, y = make_circles(n_samples=1000,
...           random_state=123, noise=0.1, factor=0.2)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...           color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...           color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

Di nuovo, supponiamo di avere un problema a due classi, dove le forme a triangolo rappresentano una classe e le forme a cerchio rappresentano un'altra classe (Figura 5.15).

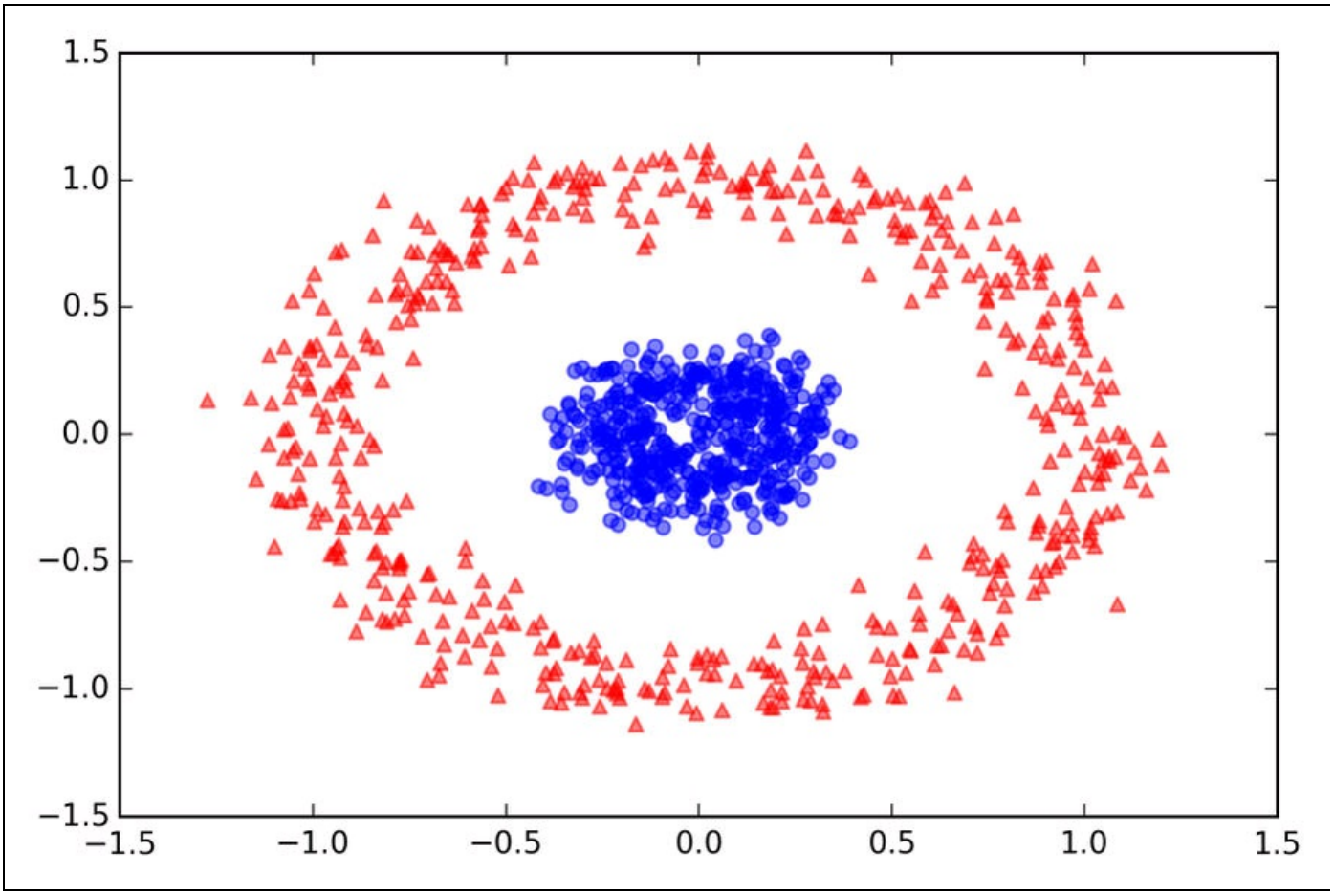


Figura 5.15

Iniziamo con l'approccio PCA standard, per confrontarlo con i risultati dell'analisi PCA kernel RBF:

```
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
...              color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Di nuovo, possiamo vedere (Figura 5.16) che l'analisi PCA standard non è in grado di produrre risultati adatti per l'addestramento di un classificatore lineare.

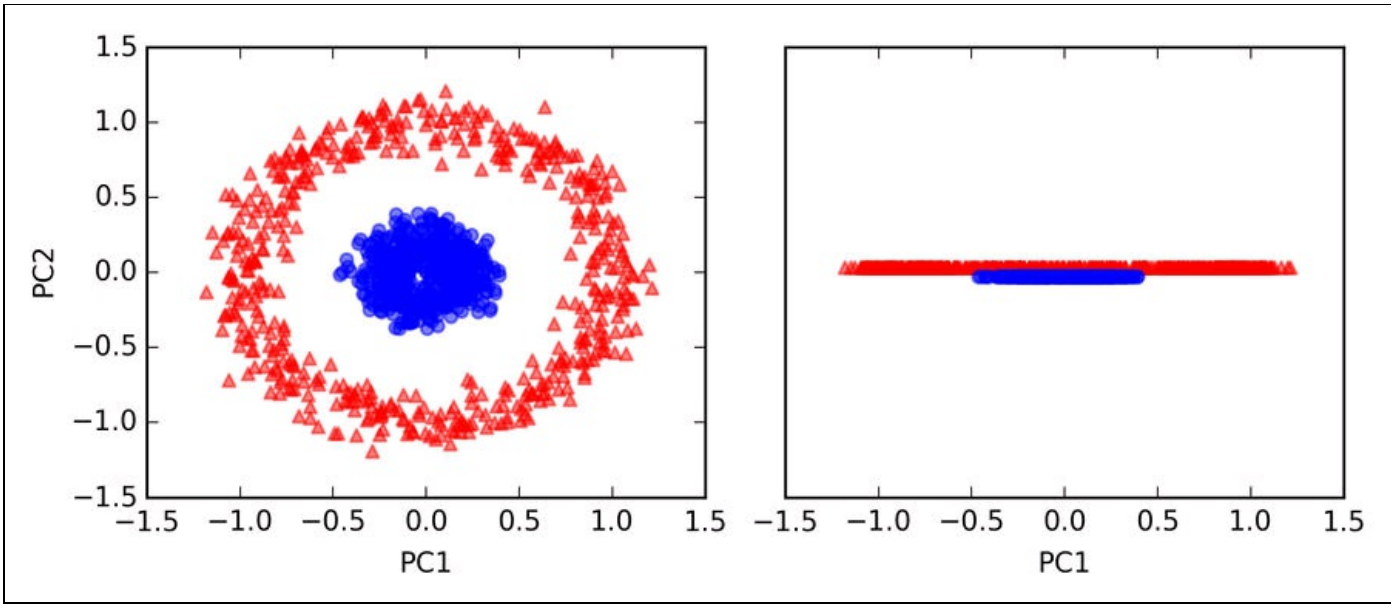


Figura 5.16

Dato un valore appropriato di γ, vediamo se siamo più fortunati utilizzando l'implementazione PCA kernel RBF:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Di nuovo, l'analisi PCA kernel RBF ha proiettato i dati su un nuovo sottospazio nel quale le due classi divengono separabili in modo lineare (Figura 5.17).

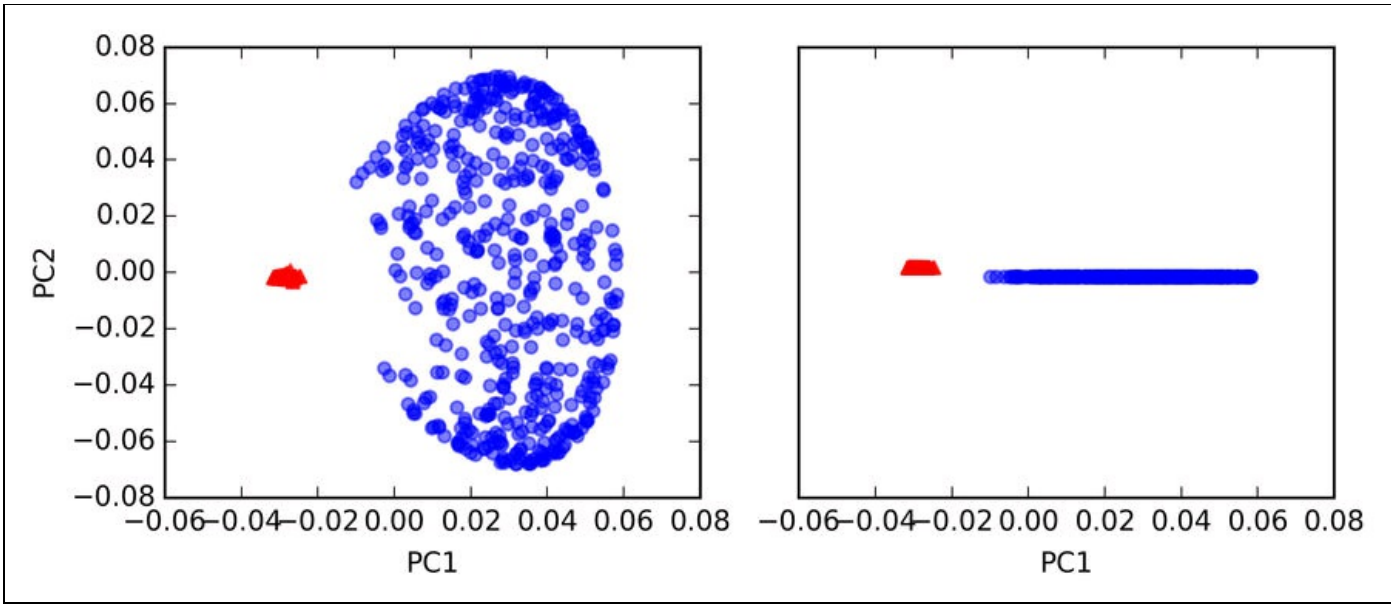


Figura 5.17

Proiezione di nuovi punti di dati

Nei due precedenti esempi di applicazione dell'analisi PCA kernel, le forme a mezzelune e a cerchi concentrici, abbiamo proiettato un singolo dataset su una nuova caratteristica. Nelle applicazioni reali, però, potremmo avere più di un dataset da trasformare, per esempio i dati di addestramento e di test e tipicamente anche i nuovi campioni, che raccoglieremo dopo la creazione e la valutazione del modello. In questo paragrafo vedremo come proiettare sul nuovo spazio i punti di dati che non facevano parte del dataset di addestramento.

Come ricorderemo dall'approccio PCA standard descritto all'inizio di questo capitolo, proiettiamo i dati calcolando il prodotto fra una matrice di trasformazione e i campioni di input; le colonne della matrice di proiezione sono i primi k autovettori ($\boldsymbol{v}$) che abbiamo ottenuto dalla matrice di covarianza. Ora, la domanda è come possiamo trasferire questo concetto all'analisi PCA kernel? Se riflettiamo sull'idea su cui si basa l'analisi PCA kernel, ricorderemo che abbiamo ottenuto un autovettore ($\boldsymbol{a}$) della matrice kernel centrata (non la matrice di covarianza), il che significa che questi sono i campioni che sono già proiettati sull'asse del componente principale $\boldsymbol{v}$. Pertanto, se vogliamo proiettare su questo asse del componente principale un nuovo campione $\boldsymbol{x}'$, dovremo calcolare il seguente:

$$\phi\left(\boldsymbol{x}'\right)^{T} \boldsymbol{v}$$

Fortunatamente, possiamo utilizzare la tecnica kernel in modo da non dover calcolare esplicitamente la proiezione

$$\phi\left(\boldsymbol{x}'\right)^T \boldsymbol{v}$$

Tuttavia, vale la pena di notare che l'analisi PCA kernel, al contrario dell'analisi PCA standard, è un metodo basato sulla memorizzazione, il che significa che, per proiettare i nuovi campioni, dobbiamo riutilizzare ogni volta il set di addestramento originario. Dobbiamo calcolare la similarità (l'accoppiamento kernel RBF) fra ogni i-esimo campione del dataset di addestramento e il nuovo campione $\boldsymbol{x}'$

$$\phi\left(\boldsymbol{x}'\right)^T \boldsymbol{v} = \sum_i a^{(i)} \phi\left(\boldsymbol{x}'\right)^T \phi\left(\boldsymbol{x}^{(i)}\right)$$

$$= \sum_i a^{(i)} k\left(\boldsymbol{x}', \boldsymbol{x}^{(i)}\right)^T$$

Qui, gli autovettori $\boldsymbol{a}$ e gli autovalori λ della matrice kernel $\boldsymbol{K}$ soddisfano la seguente condizione, nell'equazione:

$$\boldsymbol{K}\boldsymbol{a} = \lambda \boldsymbol{a}$$

Dopo aver calcolato la similarità fra i nuovi campioni e i campioni del set di addestramento, dobbiamo normalizzare l'autovettore $\boldsymbol{a}$ sulla base del suo autovalore. Pertanto, modifichiamo la funzione rbf_kernel_pca che abbiamo implementato in precedenza, in modo che restituisca anche gli autovalori della matrice kernel:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_samples, n_features]
    gamma: float
      Tuning parameter of the RBF kernel
    n_components: int
      Number of principal components to return

    Returns
    ------------
     X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
       Projected dataset
     lambdas: list
       Eigenvalues

    """
```

```
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)
    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtaining eigenpairs from the centered kernel matrix
    # numpy.eigh returns them in sorted order
    eigvals, eigvecs = eigh(K)

    # Collect the top k eigenvectors (projected samples)
    alphas = np.column_stack((eigvecs[:,-i]
                for i in range(1,n_components+1)))

    # Collect the corresponding eigenvalues
    lambdas = [eigvals[-i] for i in range(1,n_components+1)]

    return alphas, lambdas
```

Ora creiamo un nuovo dataset a mezzaluna e proiettiamolo su un sottospazio monodimensionale utilizzando l'implementazione PCA kernel RBF aggiornata:

```
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> alphas, lambdas =rbf_kernel_pca(X, gamma=15, n_components=1)
```

Per assicurarci di aver implementato il codice per la proiezione di nuovi campioni, supponiamo che il ventiseiesimo punto del dataset a mezzaluna sia un nuovo punto dati x' e che il nostro compito sia di proiettarlo su questo nuovo sottospazio:

```
>>> x_new = X[25]
>>> x_new
array([ 1.8713187 ,  0.00928245])
>>> x_proj = alphas[25] # original projection
>>> x_proj
array([ 0.07877284])
>>> def project_x(x_new, X, gamma, alphas, lambdas):
...     pair_dist = np.array([np.sum(
...                 (x_new-row)**2) for row in X])
...     k = np.exp(-gamma * pair_dist)
... return k.dot(alphas / lambdas)
```

Tramite il codice seguente, siamo in grado di riprodurre la proiezione originale. Utilizzando la funzione `project_x` saremo in grado di proiettare altrettanto bene ogni nuovo campione. Il codice è:

```
>>> x_reproj = project_x(x_new, X,
...       gamma=15, alphas=alphas, lambdas=lambdas)
>>> x_reproj
array([ 0.07877284])
```

Infine, visualizziamo la proiezione sul primo componente principale:

```
>>> plt.scatter(alphas[y==0, 0], np.zeros((50)),
...             color='red', marker='^',alpha=0.5)
>>> plt.scatter(alphas[y==1, 0], np.zeros((50)),
...             color='blue', marker='o', alpha=0.5)
>>> plt.scatter(x_proj, 0, color='black',
...             label='original projection of point X[25]',
...             marker='^', s=100)
>>> plt.scatter(x_reproj, 0, color='green',
...             label='remapped point X[25]',
```

```
...             marker='x', s=500)
>>> plt.legend(scatterpoints=1)
>>> plt.show()
```

Come possiamo vedere nel grafico a dispersione rappresentato nella Figura 5.18, abbiamo mappato correttamente il campione x' sul primo componente principale.

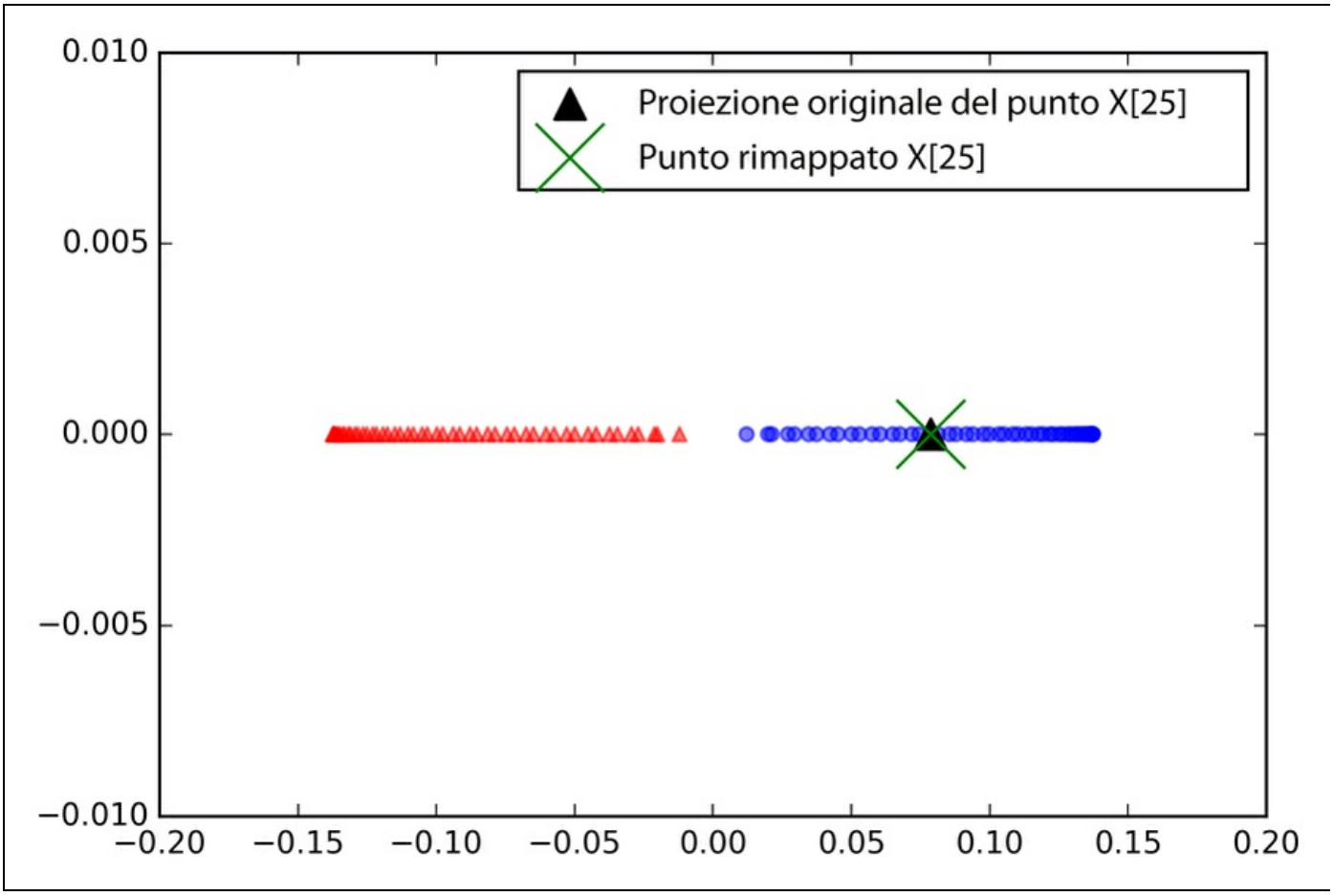


Figura 5.18

Kernel PCA con scikit-learn

Per comodità, scikit-learn implementa una classe PCA nel modulo `sklearn.decomposition`. L'uso è simile alla classe PCA standard e possiamo specificare il kernel tramite il parametro `kernel`:

```
>>> from sklearn.decomposition import KernelPCA
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> scikit_kpca = KernelPCA(n_components=2,
...             kernel='rbf', gamma=15)
>>> X_skernpca = scikit_kpca.fit_transform(X)
```

Per vedere se otteniamo risultati coerenti con l'implementazione PCA kernel, tracciamo il grafico delle forme a mezzaluna, trasformate sui due componenti principali:

```
>>> plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
... color='red', marker='^', alpha=0.5)
>>> plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
... color='blue', marker='o', alpha=0.5)
```

```
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.show()
```

Come possiamo vedere nella Figura 5.19, i risultati di KernelPCA di scikit-learn sono coerenti con la nostra implementazione.

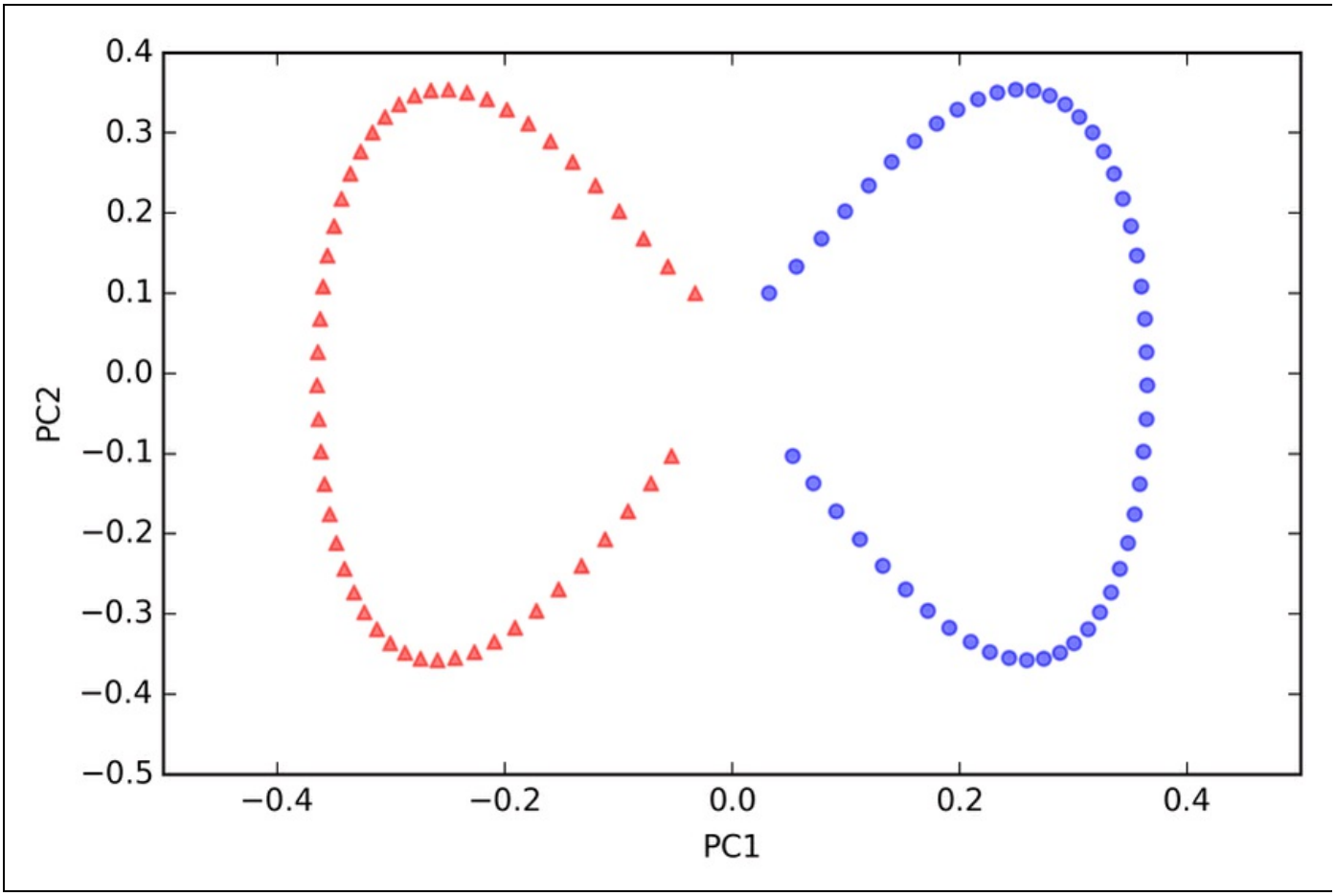


Figura 5.19

NOTA

Scikit-learn implementa anche alcune tecniche avanzate per la riduzione della dimensionalità non lineare, che però non rientrano negli scopi di questo libro. Un'ottima panoramica sulle implementazioni correnti presenti in scikit-learn ricca di esempi illustrativi si trova in http://scikit-learn.org/stable/modules/manifold.html.

Riepilogo

In questo capitolo abbiamo trattato tre diversi tipi d'analisi, sostanzialmente tecniche di riduzione della dimensionalità per l'estrazione delle caratteristiche: PCA standard, LDA e kernel PCA. Utilizzando PCA, abbiamo proiettato i dati su un sottospazio di minori dimensioni, in modo da massimizzare la varianza fra gli assi ortogonali delle caratteristiche, ignorando le etichette delle classi. LDA, al contrario di PCA, è una tecnica per la riduzione della dimensionalità con supervisione, ovvero considerando le informazioni sulla classe già presenti nel dataset di addestramento, per tentare di massimizzare la separabilità in classi in uno spazio di caratteristiche lineari. Infine abbiamo parlato della versione kernel di PCA, che consente di mappare dataset non lineari su uno spazio delle caratteristiche dotato di dimensionalità inferiore, dove le classi possono risultare separabili in modo lineare.

Grazie a queste tecniche di pre-elaborazione, siamo pronti per conoscere i migliori metodi per incorporare in modo efficiente le varie tecniche di pre-elaborazione e valutare le prestazioni dei vari modelli, argomento del prossimo capitolo.

Valutazione dei modelli e ottimizzazione degli iperparametri

Nei capitoli precedenti abbiamo parlato dei principali algoritmi di machine learning per la classificazione e di come sia opportuno dare un aspetto appropriato ai dati, prima di inviarli a questi algoritmi. Ora è il momento di conoscere le migliori tecniche per la realizzazione di buoni modelli di machine learning, ottimizzando gli algoritmi e valutando le prestazioni risultanti nel modello. In questo capitolo parleremo dei seguenti argomenti.

- Ottenere stime non distorte delle prestazioni di un modello.
- Individuare i problemi comuni degli algoritmi di machine learning.
- Ottimizzare i modelli di machine learning.
- Valutare i modelli predittivi, utilizzando differenti metriche prestazionali.

Accelerare il flusso di lavoro

Quando abbiamo applicato le varie tecniche di pre-elaborazione descritte nei capitoli precedenti, come la standardizzazione delle caratteristiche e la loro riduzione in scala, nel Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione* o l'*analisi dei componenti principali* e la compressione dei dati nel Capitolo 5, *Compressione dei dati tramite la riduzione della dimensionalità*, abbiamo scoperto che dovevamo riutilizzare i parametri che avevamo ottenuto durante l'adattamento dei dati di addestramento e riapplicarli per ridurre in scala e comprimere ogni nuovo dato, per esempio i campioni presenti nel dataset di test. In questo paragrafo parleremo di uno strumento molto comodo, la classe `Pipeline` di scikit-learn. Questo ci consente di adattare un modello includendo un numero arbitrario di passi di trasformazione, e applicarlo poi per eseguire previsioni sui nuovi dati in arrivo.

Caricamento del dataset relativo al cancro al seno nel Wisconsin

In questo capitolo lavoreremo sul dataset *Breast Cancer Wisconsin*, che contiene 569 campioni di celle tumorali maligne e benigne. Le prime due colonne del dataset conservano il codice numerico univoco dei campioni e la diagnosi corrispondente (*M=maligno*, *B=benigno*). Le colonne da 3 a 32 contengono trenta caratteristiche, sotto forma di valori effettivi, che sono state calcolate sulla base delle immagini digitalizzate del nucleo della cellula, e che possono essere utilizzate per costruire un modello di previsione del fatto che un tumore sia benigno o maligno. Il dataset Breast Cancer Wisconsin è stato depositato nell'*UCI machine learning repository* e informazioni dettagliate su questo dataset si possono trovare in https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic).

In questo paragrafo leggeremo il dataset e lo suddivideremo in dataset di addestramento e di test tramite tre semplici passi.

1. Inizieremo leggendo il dataset direttamente dal sito web UCI utilizzando `pandas`:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data', header=None)
```

2. Poi assegneremo le trenta caratteristiche a un array NumPy di nome `X`. Utilizzando `LabelEncoder` trasformeremo le etichette delle classi dalla loro originale

rappresentazione sotto forma di stringhe (M e B) in valori interi:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> X = df.loc[:, 2:].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

Dopo aver codificato le etichette delle classi (diagnosi) in un array `y`, ai tumori maligni viene assegnata la classe 1 e ai tumori benigni la classe 0, cosa che possiamo illustrare richiamando il metodo `transform` di `LabelEncoder` su due etichette delle classi fittizie:

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

3. Prima di costruire la nostra prima pipeline del modello nei prossimi paragrafi, suddividiamo il dataset in un dataset di addestramento (80% dei dati) e un dataset di test (20% dei dati):

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = \
...      train_test_split(X, y, test_size=0.20, random_state=1)
```

Combinare i trasformatori e gli estimatori in una pipeline

Nel capitolo precedente abbiamo visto che molti algoritmi di apprendimento richiedono che le caratteristiche di input si trovino sulla stessa scala per garantire prestazioni ottimali. Pertanto, dobbiamo standardizzare le colonne della classe *Breast Cancer Wisconsin* prima di poterle inviare al classificatore lineare, come la regressione logistica. Inoltre, supponiamo di voler comprimere i nostri dati dalle 30 dimensioni iniziali a un sottospazio più compatto, bidimensionale, tramite l'analisi *PCA* (*Principal Component Analysis*), una tecnica di riduzione della dimensionalità di cui abbiamo parlato nel Capitolo 5, *Compressione dei dati tramite la riduzione della dimensionalità*. Invece di svolgere i passi di adattamento e trasformazione in modo separato per i dataset di addestramento e di test, possiamo concatenare gli oggetti `StandardScaler`, `PCA` e `LogisticRegression` in una pipeline:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import Pipeline
>>> pipe_lr = Pipeline([('scl', StandardScaler()),
...           ('pca', PCA(n_components=2)),
...           ('clf', LogisticRegression(random_state=1))])
>>> pipe_lr.fit(X_train, y_train)
>>> print('Test Accuracy: %.3f' % pipe_lr.score(X_test, y_test))
Test Accuracy: 0.947
```

L’oggetto Pipeline accetta in input un elenco di tuple, dove il primo valore di ciascuna tupla è un identificatore stringa arbitrario, che possiamo utilizzare per accedere ai singoli elementi della pipeline, come vedremo più avanti in questo capitolo, e il secondo elemento di ogni tupla è un trasformatore o estimatore scikit-learn.

I passi intermedi di una pipeline costituiscono i trasformatori scikit-learn e l’ultimo passo è un estimatore. Nell’esempio di codice precedente, abbiamo costruito una pipeline che era costituita da due passi intermedi, uno StandardScaler e un trasformatore PCA e un classificatore a regressione logistica quale estimatore finale. Quando abbiamo eseguito il metodo fit sulla pipeline pipe_lr, lo StandardScaler ha eseguito fit e transform sui dati di addestramento; i dati di addestramento trasformati sono stati poi passati all’oggetto successivo della pipeline, PCA. Come nei passi precedenti, anche PCA ha eseguito fit e transform sui dati di input in scala e li ha passati all’elemento finale della pipeline, l’estimatore. Occorre notare che non vi è alcun limite al numero di passi intermedi di questa pipeline. Il funzionamento della pipeline è illustrato nella Figura 6.1.

Uso della convalida incrociata k-fold per valutare le prestazioni del modello

Uno dei passi chiave nella costruzione di un modello di apprendimento consiste nello stimare le sue prestazioni sui dati che il modello non ha mai visto prima. Supponiamo di adattare il nostro modello su un dataset di addestramento e di utilizzare gli stessi dati per stimare come si comporti in pratica.

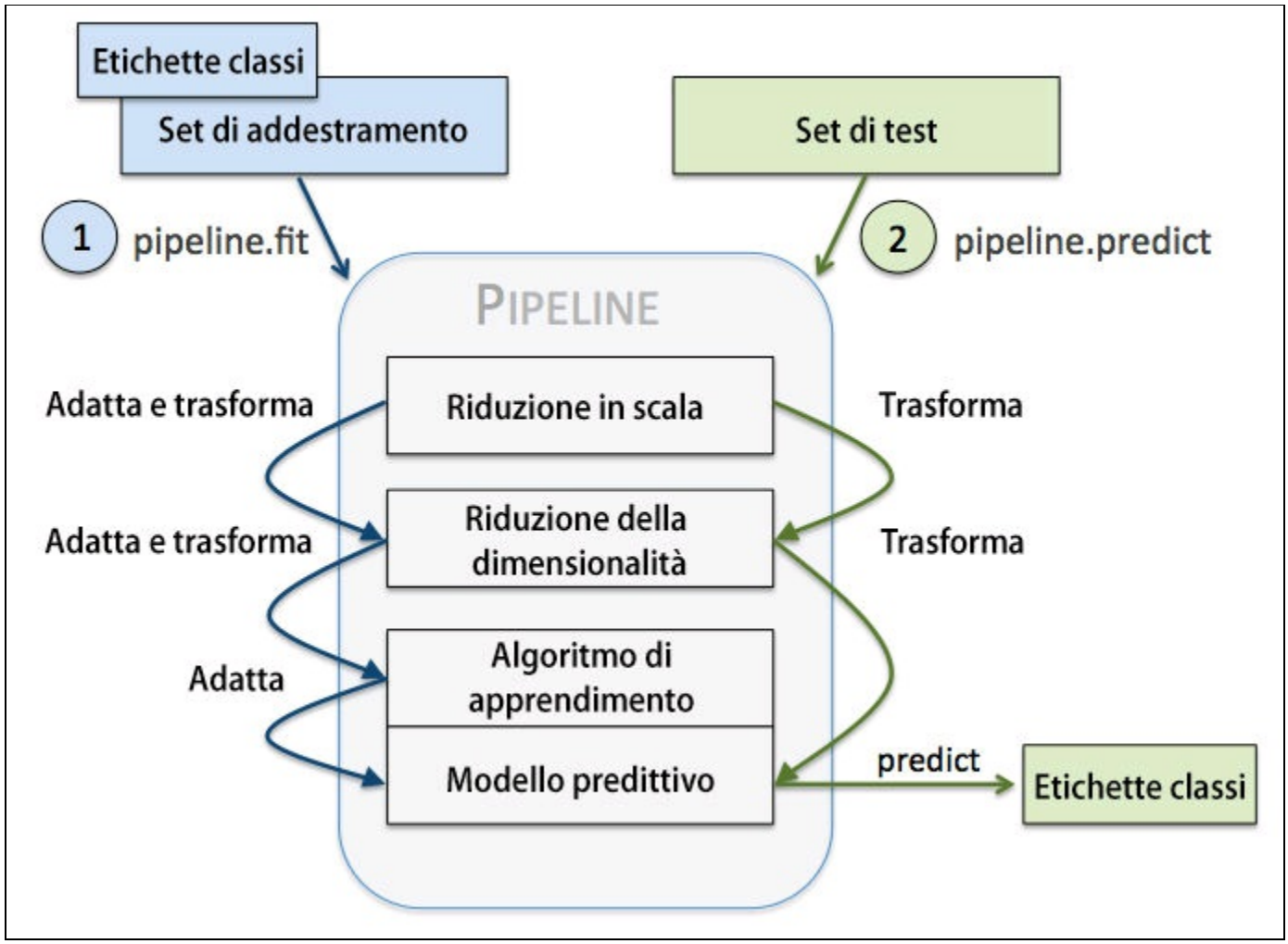


Figura 6.1 Il funzionamento di una pipeline.

Possiamo ricordare dal paragrafo *Risolvere l'overfitting tramite la regolarizzazione* del Capitolo 3, *I classificatori di machine learning di scikit-learn*, che un modello può soffrire di underfitting (elevato bias), se il modello è troppo semplice, o di overfitting dei dati di addestramento (elevata varianza) se il modello è troppo complesso per i dati di addestramento sottostanti. Per trovare un compromesso accettabile bias-varianza, dobbiamo valutare con cura il nostro modello. In questo paragrafo esamineremo alcune utili tecniche di convalida

incrociata, la *convalida incrociata holdout* e la *convalida incrociata k-fold*, che possono aiutarci a ottenere stime affidabili dell'errore di generalizzazione del modello, ovvero di come si comporta su dati mai visti prima.

Il modello holdout

Un approccio classico e molto utilizzato per la stima delle prestazioni di generalizzazione dei modelli di machine learning è la convalida incrociata holdout. Utilizzando il metodo holdout, suddividiamo il nostro dataset iniziale in un dataset di addestramento e un dataset distinto di test. Il primo viene utilizzato per l'addestramento del modello e il secondo per la stima delle sue prestazioni. Tuttavia, in una tipica applicazione di machine learning, siamo anche interessati a ottimizzare e confrontare impostazioni differenti dei parametri, per migliorare ulteriormente le prestazioni nell'effettuare previsioni su dati mai visti. Questo processo è chiamato *selezione del modello*, dove il termine fa riferimento a un determinato problema di classificazione per il quale vogliamo selezionare valori ottimali dei parametri di configurazione (gli *iperparametri*). Tuttavia, se riutilizziamo più volte lo stesso dataset di test durante la selezione del modello, questo entrerà a far parte dei nostri dati di addestramento e pertanto il modello sarà maggiormente soggetto a overfit. Nonostante questo problema, molti usano ancora il set di test per la selezione del modello, il che non è una buona pratica per la scelta dell'algoritmo di machine learning.

Un modo migliore per utilizzare il metodo holdout per la selezione del modello consiste nel separare i dati in tre parti: un set di addestramento, un set di convalida e un set di test. Il set di addestramento viene utilizzato per adattare diversi modelli e le prestazioni sul set di convalida vengono poi utilizzate per la selezione del modello. Il vantaggio di avere un set di test che il modello non ha visto prima durante l'addestramento e per i passi di selezione del modello è il fatto che possiamo ottenere una stima meno distorta sulla sua capacità di generalizzazione sui nuovi dati. La Figura 6.2 illustra il concetto di convalida incrociata holdout, dove possiamo utilizzare un set di convalida per valutare ripetutamente le prestazioni del modello dopo aver averlo addestrato utilizzando valori differenti dei parametri. Una volta che siamo soddisfatti dei parametri, possiamo stimare l'errore di generalizzazione dei modelli sul dataset di test.

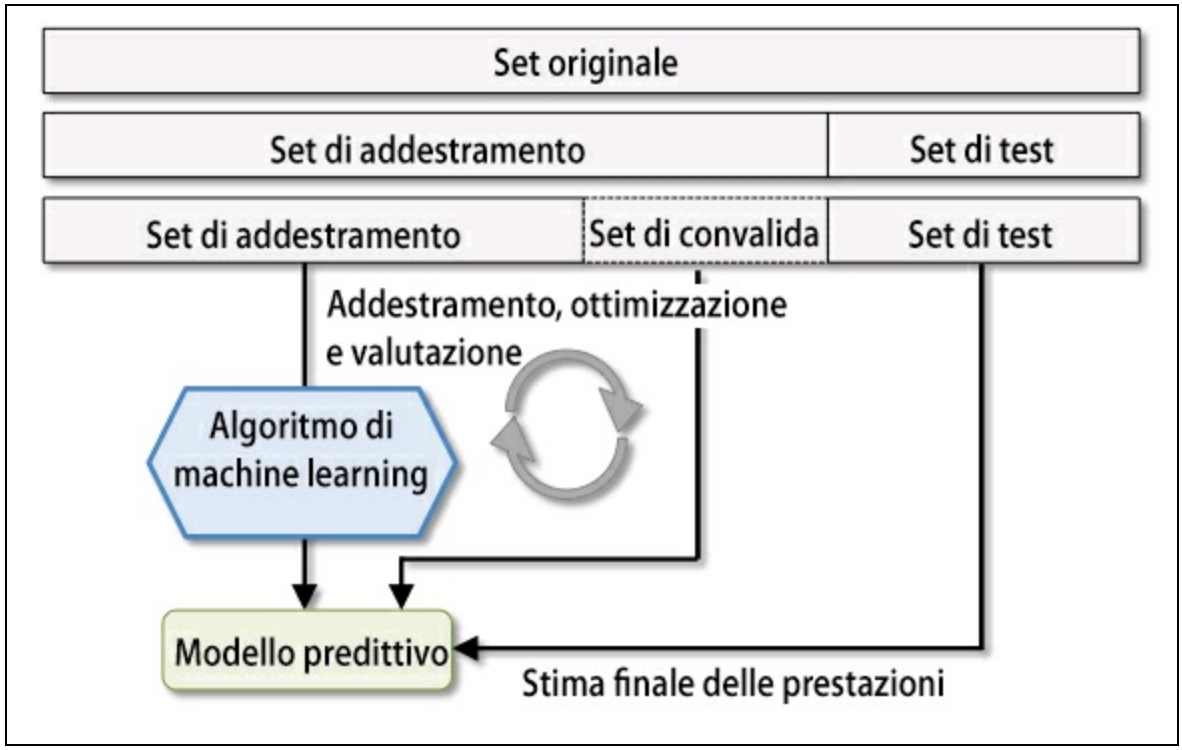


Figura 6.2

Uno svantaggio del metodo holdout è il fatto che la stima delle prestazioni è sensibile al modo in cui suddividiamo il set di addestramento fra i sottoinsiemi di addestramento e convalida; la stima varierà per campioni di dati differenti. Nel prossimo paragrafo esamineremo una tecnica migliore per la stima delle prestazioni, la convalida incrociata k-fold, dove possiamo ripetere il metodo holdout k volte su k sottoinsiemi dei dati di addestramento.

La convalida incrociata K-fold

Nella convalida incrociata K-fold, suddividiamo in modo casuale il dataset di addestramento in k parti senza reinserimento: $k-1$ parti vengono utilizzate per l'addestramento del modello e una parte viene utilizzata per il collaudo. Questa procedura viene ripetuta k volte in modo da ottenere k modelli e stime di prestazioni.

Approfondimento

Nel caso non abbiate familiarità con i termini di campionamento *con* e *senza* reinserimento, svolgiamo un semplice esperimento concettuale. Supponiamo di giocare alla lotteria dove estraiamo casualmente dei numeri da un'urna. Cominciamo con un'urna che contiene cinque numeri univoci, 0, 1, 2, 3 e 4, ed estraiamo esattamente un numero a ciascun turno. Al primo turno, le probabilità di estrarre un determinato numero saranno esattamente di 1/5. Ora, nel campionamento senza reinserimento, non ricollochiamo il numero nell'urna dopo ciascun turno. Di conseguenza, la probabilità di estrarre un determinato numero dal set dei numeri rimanenti nel

turno successivo dipende dal turno precedente. Per esempio, se il set rimanente contiene i numeri 0, 1, 2 e 4, la probabilità di estrarre il numero 0, al turno successivo, sarà pari a 1/4.
Al contrario, nel campionamento casuale con reinserimento, rimettiamo sempre il numero estratto nell'urna, in modo che le probabilità di estrarre un determinato numero a ciascun turno non cambi; possiamo estrarre lo stesso numero più di una volta. In altre parole, nel campionamento con reinserimento, i campioni (numeri) sono indipendenti e hanno covarianza zero. Per esempio, i risultati dei cinque turni dell'estrazione dei numeri casuali potrebbero essere i seguenti.

- Campionamento casuale con reinserimento: 2, 1, 3, 4, 0
- Campionamento casuale senza reinserimento: 1, 3, 3, 4, 1

Poi calcoliamo le prestazioni medie dei modelli sulla base delle suddivisioni differenti indipendenti, per ottenere una stima delle prestazioni che sia meno sensibile al partizionamento dei dati di addestramento rispetto al metodo holdout. Tipicamente, utilizziamo la convalida incrociata k-fold per l'ottimizzazione del modello, ovvero per trovare i valori ottimali degli iperparametri che forniscono prestazioni di generalizzazione soddisfacenti. Una volta che abbiamo trovato i valori soddisfacenti degli iperparametri, possiamo riaddestrare il modello sull'intero set di e ottenere una stima finale delle prestazioni utilizzando il set di test indipendente.

Poiché la convalida incrociata k-fold è una tecnica di ricampionamento senza reinserimento, il vantaggio di questo approccio è che ciascun punto campione farà parte dei dataset di addestramento e di test esattamente una volta, il che fornisce una stima a varianza più bassa delle prestazioni del modello rispetto al metodo holdout. La Figura 6.3 riepiloga il concetto su cui si basa la convalida incrociata k-fold con $k = 10$. Il dataset di addestramento viene diviso in dieci parti e durante le dieci iterazioni, nove parti vengono utilizzate per l'addestramento e una parte viene utilizzata quale set di test per la valutazione del modello. Inoltre, le prestazioni stimate E_i (per esempio l'accuratezza della classificazione o l'errore) di ciascuna parte vengono poi utilizzate per calcolare le prestazioni stimate medie E del modello.

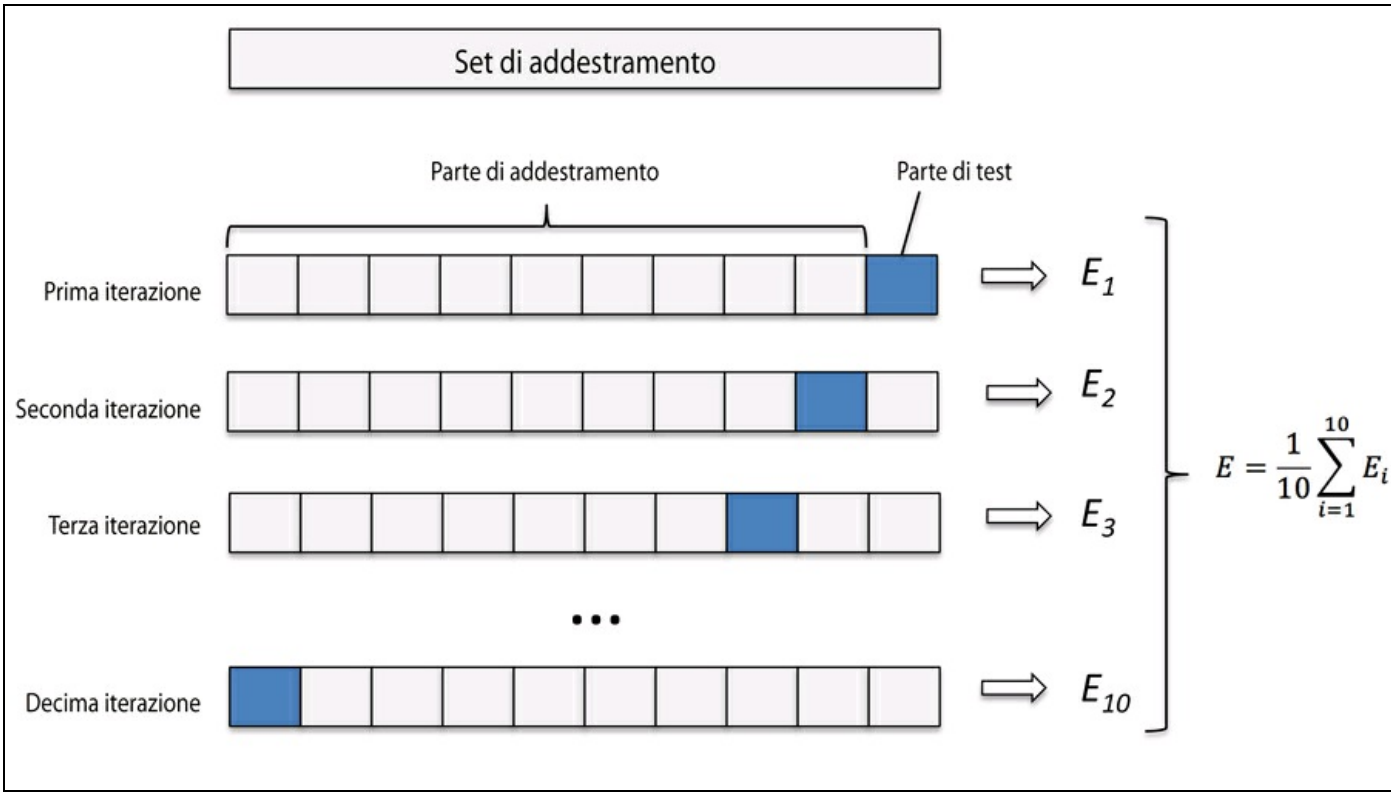


Figura 6.3

Il valore standard di *k* per la convalida incrociata k-fold è pari a 10, che è ragionevole per la maggior parte delle applicazioni. Tuttavia, se utilizziamo set di addestramento relativamente ridotti, sarebbe utile incrementare il numero di parti. Se incrementiamo il valore di *k*, a ogni iterazione verranno utilizzati più dati di addestramento, il che produrrà un bias inferiore nelle stime delle prestazioni di generalizzazione sulla media delle singole stime dei modelli. Tuttavia, valori troppo ampi di *k* aumenteranno anche il tempo di esecuzione dell'algoritmo di convalida incrociata, fornendo nel contempo stime con una varianza più elevata, in quanto le parti di addestramento saranno più simili le une alle altre. E d'altra parte, se stiamo lavorando su dataset di grandi dimensioni, possiamo scegliere un valore di *k* più ridotto, per esempio $k = 5$, e ottenere comunque una stima accurata delle prestazioni medie del modello, riducendo il costo computazionale del riadattamento della valutazione del modello per le varie parti.

NOTA

Un caso speciale di convalida incrociata k-fold si chiama *LOO (Leave-One-Out)*. Nella convalida incrociata LOO, impostiamo il numero di parti uguale al numero di campioni di addestramento ($k = n$), in modo che a ogni iterazione venga utilizzato per il collaudo un solo campione di addestramento. Questo è un approccio consigliato per dataset molto piccoli.

Un leggero miglioramento rispetto all'approccio a convalida incrociata k-fold è la convalida incrociata k-fold stratificata, che può fornire migliori stime di bias e

varianza, specialmente nei casi di proporzioni differenti fra le classi, come si può vedere in uno studio di R. Kohavi (R. Kohavi et al. *A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection*, in "Ijcai", volume 14, pp. 1137–1145, 1995). Nella convalida incrociata stratificata, le proporzioni fra le classi vengono preservate in ciascuna parte, per garantire che ciascuna parte sia rappresentativa delle proporzioni fra le classi del dataset di addestramento, che illustreremo utilizzando l'iteratore `StratifiedKFold` di scikit-learn:

```
>>> import numpy as np
>>> from sklearn.cross_validation import StratifiedKFold
>>> kfold = StratifiedKFold(y=y_train,
...                         n_folds=10,
...                         random_state=1)
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print('Fold: %s, Class dist.: %s, Acc: %.3f' % (k+1,
...           np.bincount(y_train[train]), score))
Fold: 1, Class dist.: [256 153], Acc: 0.891
Fold: 2, Class dist.: [256 153], Acc: 0.978
Fold: 3, Class dist.: [256 153], Acc: 0.978
Fold: 4, Class dist.: [256 153], Acc: 0.913
Fold: 5, Class dist.: [256 153], Acc: 0.935
Fold: 6, Class dist.: [257 153], Acc: 0.978
Fold: 7, Class dist.: [257 153], Acc: 0.933
Fold: 8, Class dist.: [257 153], Acc: 0.956
Fold: 9, Class dist.: [257 153], Acc: 0.978
Fold: 10, Class dist.: [257 153], Acc: 0.956
>>> print('CV accuracy: %.3f +/- %.3f' % (
...       np.mean(scores), np.std(scores)))
CV accuracy: 0.950 +/- 0.029
```

Innanzitutto abbiamo inizializzato l'iteratore `StratifiedKfold` dal modulo `sklearn.cross_validation` con le etichette delle classi `y_train` del set di addestramento e abbiamo specificato il numero di parti tramite il parametro `n_folds`. Quando abbiamo utilizzato l'iteratore `kfold` per esaminare le `k` parti, abbiamo utilizzato gli indici `train` per adattare la pipeline a regressione logistica che abbiamo configurato all'inizio di questo capitolo. Utilizzando la pipeline `pile_lr`, abbiamo garantito che a ogni iterazione i campioni fossero nella scala corretta (per esempio standardizzati). Poi abbiamo utilizzato gli indici `test` per calcolare il punteggio di accuratezza del modello, che abbiamo raccolto nella lista `scores` per calcolare l'accuratezza media e la deviazione standard della stima.

Sebbene il precedente esempio di codice fosse utile per illustrare come funziona la convalida incrociata k-fold, scikit-learn implementa anche un valutatore della convalida incrociata k-fold, che ci offre anche la possibilità di valutare il nostro modello utilizzando una più efficiente la convalida incrociata k-fold stratificata:

```
>>> from sklearn.cross_validation import cross_val_score
>>> scores = cross_val_score(estimator=pipe_lr,
...                          X=X_train,
...                          y=y_train,
```

```
...                     cv=10,
...                     n_jobs=1)
>>> print('CV accuracy scores: %s' % scores)
CV accuracy scores: [ 0.89130435  0.97826087  0.97826087
                0.91304348  0.93478261  0.97777778
                0.93333333  0.95555556  0.97777778
                  0.95555556]
>>> print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
CV accuracy: 0.950 +/- 0.029
```

Una caratteristica estremamente utile dell'approccio `cross_val_score` è il fatto che possiamo distribuire la valutazione delle varie parti su più CPU della nostra macchina. Se impostiamo il parametro `n_jobs` a `1`, solo una CPU verrà utilizzata per valutare le prestazioni, come avviene nell'esempio `StratifiedKFold` precedente. Tuttavia, se impostiamo `n_jobs=2`, possiamo distribuire i dieci turni di convalida incrociata su due CPU (se sono disponibili nella macchina) e impostando `n_jobs=-1`, possiamo utilizzare tutte le CPU disponibili della macchina, che eseguiranno il calcolo in parallelo.

NOTA

Una discussione dettagliata sul modo in cui la varianza delle prestazioni di generalizzazione viene stimata nella convalida incrociata non può rientrare negli scopi di questo libro, ma potete trovare una discussione dettagliata in un eccellente articolo di M. Markatou (M. Markatou, H. Tian, S. Biswas e G. M. Hripcsak, *Analysis of Variance of Cross-validation Estimators of the Generalization Error, in "Journal of Machine Learning Research"*, 6:1127–1168, 2005).

Esistono anche tecniche alternative di convalida incrociata, fra cui il metodo della convalida incrociata .632 Bootstrap (B. Efron e R. Tibshirani, *Improvements on Cross-validation: The 632+ Bootstrap Method*, in "Journal of the American Statistical Association", 92(438):548–560, 1997).

Debugging degli algoritmi con le curve di apprendimento e di convalida

Ora esamineremo due semplici, ma potenti, strumenti diagnostici che possono aiutarci a migliorare le prestazioni dell'algoritmo di apprendimento: le *curve di apprendimento* e le *curve di convalida*. Poi, nei paragrafi successivi, vedremo come possiamo utilizzare le curve di apprendimento per diagnosticare se un algoritmo di apprendimento ha un problema di overfitting (elevata varianza) o underfitting (elevato bias). Inoltre, esamineremo le curve di convalida, che possono aiutarci a risolvere i più comuni problemi degli algoritmi di apprendimento.

Diagnosi dei problemi di bias e varianza con le curve di apprendimento

Se un modello è troppo complesso per un determinato dataset di addestramento (offretroppi gradi di libertà o parametri), il modello tende a non convergere a sufficienza (overfit) sui dati di addestramento e a non essere ben generalizzabile su dati mai visti. Talvolta basta raccogliere più campioni di addestramento per ridurre il grado di overfitting. Tuttavia, all'atto pratico, spesso può essere molto costoso o addirittura impossibile raccogliere ulteriori dati. Tracciando la precisione di addestramento e convalida del modello come funzioni delle dimensioni del set di addestramento, possiamo individuare con maggiore facilità se il modello soffre di elevata varianza o elevato bias e se la raccolta di ulteriori dati possa aiutare a risolvere questo problema. Ma prima di trattare il modo in cui tracciare queste curve in scikit-learn, parliamo dei due problemi più comuni dei modelli, sulla base della Figura 6.4.

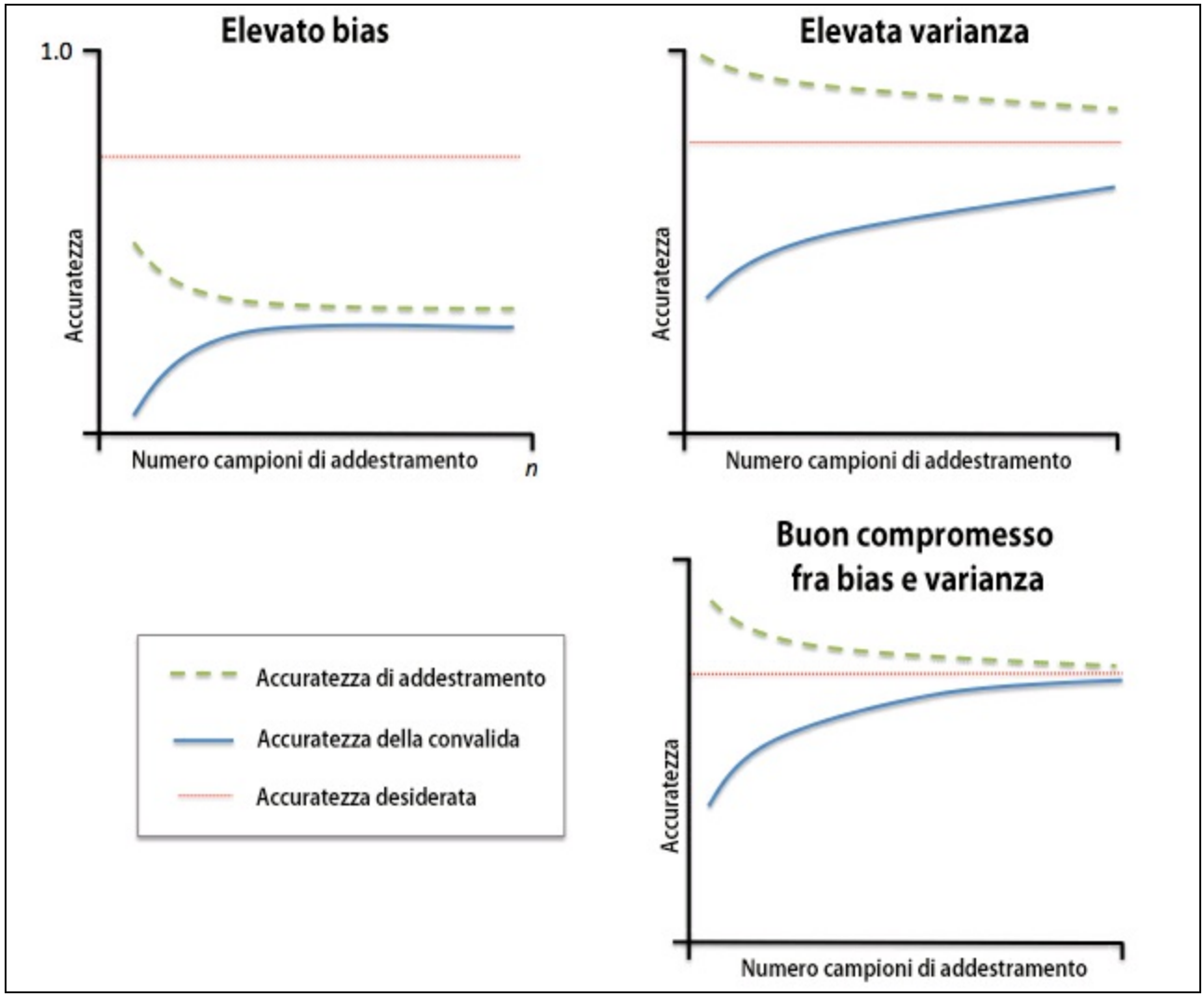


Figura 6.4

Il grafico in alto a sinistra mostra un modello con elevato bias. Questo modello ha una scarsa accuratezza sia di addestramento sia di convalida incrociata: in pratica ha ha problemi di underfit sui dati di addestramento. Per risolvere questo problema, in genere si aumenta il numero di parametri del modello, per esempio raccogliendo o costruendo ulteriori caratteristiche e riducendo il grado di regolarizzazione, per esempio nei classificatori SVM o a regressione logistica. Il grafico in alto a destra mostra un modello che soffre di elevata varianza, il che è indicato dalla distanza troppo ampia fra l'accuratezza di addestramento e l'accuratezza di convalida incrociata. Per risolvere questo problema (overfitting), possiamo raccogliere più dati di addestramento oppure ridurre la complessità del modello, per esempio aumentando le dimensioni del parametro di regolarizzazione; per i modelli non regolarizzati, può essere utile anche ridurre il numero delle caratteristiche, tramite la loro selezione (Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione*) o la loro estrazione (Capitolo 5, *Compressione dei dati tramite la*

riduzione della dimensionalità). Dobbiamo notare che la raccolta di una maggiore quantità di dati di addestramento riduce le probabilità di un overfitting. Tuttavia, non sempre questo potrebbe essere utile, per esempio quando i dati di addestramento sono estremamente rumorosi o il modello è già piuttosto vicino a quello ottimale.

Nei prossimi paragrafi vedremo come risolvere questi problemi dei modelli utilizzando le curve di convalida, ma prima vediamo come possiamo utilizzare la funzione della curva di apprendimento di scikit-learn per valutare il modello:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.learning_curve import learning_curve
>>> pipe_lr = Pipeline([
...          ('scl', StandardScaler()),
...          ('clf', LogisticRegression(
...                       penalty='l2', random_state=0))])
>>> train_sizes, train_scores, test_scores =\
...       learning_curve(estimator=pipe_lr,
...                      X=X_train,
...                      y=y_train,
...                      train_sizes=np.linspace(0.1, 1.0, 10),
...                      cv=10,
...                      n_jobs=1)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(train_sizes, train_mean,
...          color='blue', marker='o',
...          markersize=5,
...          label='training accuracy')
>>> plt.fill_between(train_sizes,
...                  train_mean + train_std,
...                  train_mean - train_std,
...                  alpha=0.15, color='blue')
>>> plt.plot(train_sizes, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='validation accuracy')
>>> plt.fill_between(train_sizes,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Number of training samples')
>>> plt.ylabel('Accuracy')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

Dopo aver eseguito con successo il codice precedente, otterremo la curva di apprendimento rappresentata nella Figura 6.5.

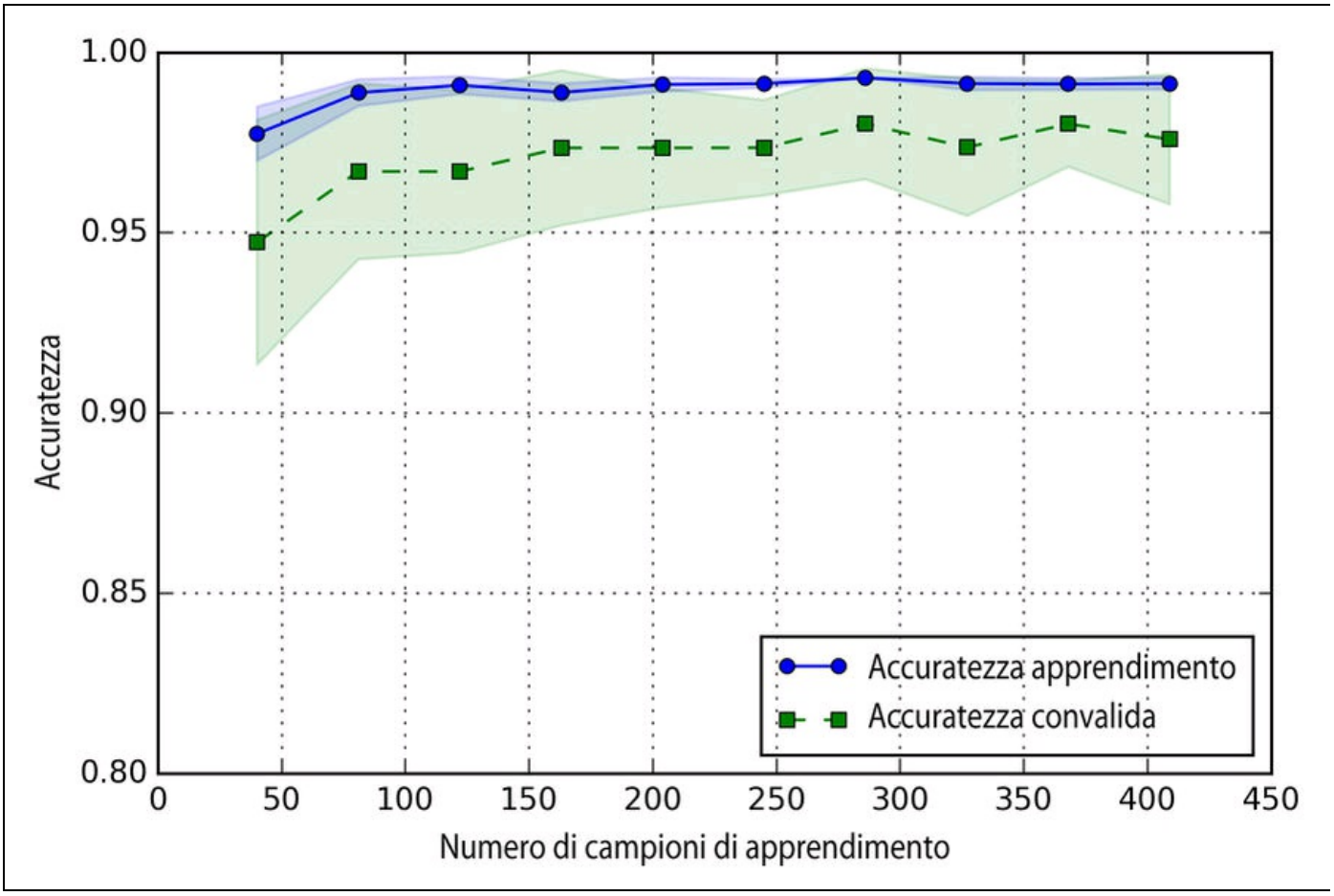


Figura 6.5

Tramite il parametro `train_sizes` della funzione `learning_curve`, possiamo controllare il numero assoluto o relativo dei campioni di apprendimento che vengono utilizzati per generare le curve di apprendimento. Qui, possiamo impostare `train_sizes=np.linspace(0.1, 1.0, 10)` per utilizzare 10 intervalli relativi uniformemente spaziati per le dimensioni del set di addestramento. Per impostazione predefinita, la funzione `learning_curve` utilizza la convalida incrociata k-fold stratificata per calcolare l'accuratezza e impostiamo $k = 10$ tramite il parametro `cv`. Poi calcoliamo semplicemente la media dell'accuratezza dall'addestramento a convalida incrociata che ci viene restituito, con le valutazioni dei test per varie dimensioni del set di addestramento, e tracciamo il tutto utilizzando la funzione `plot` di matplotlib. Inoltre, aggiungiamo al grafico la deviazione standard delle accuratezze medie, utilizzando la funzione `fill_between`, per indicare la varianza della stima.

Come possiamo vedere nella Figura 6.5 precedente, il nostro modello si comporta piuttosto bene sul dataset di test. Tuttavia, potrebbe subire un leggero overfitting sui dati di addestramento, il che è indicato da un limitato, ma pur sempre

visibile, gap fra le curve di accuratezza dell'apprendimento e della convalida incrociata.

Soluzione dei problemi di overfitting e underfitting con le curve di convalida

Le curve di convalida sono uno strumento utile per migliorare le prestazioni di un modello risolvendo i suoi problemi di overfitting o underfitting. Le curve di convalida sono legate alle curve di apprendimento, ma invece di tracciare l'accuratezza di addestramento di test in funzione delle dimensioni del campione, interveniamo sui parametri del modello, per esempio il parametro di regolarizzazione inversa `c` nella regressione logistica. Procediamo e vediamo come creare le curve di convalida tramite sckit:

```
>>> from sklearn.learning_curve import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
... estimator=pipe_lr,
... X=X_train,
... y=y_train,
... param_name='clf__C',
... param_range=param_range,
... cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(param_range, train_mean,
...         color='blue', marker='o',
...         markersize=5,
...         label='training accuracy')
>>> plt.fill_between(param_range, train_mean + train_std,
...                  train_mean - train_std, alpha=0.15,
...                  color='blue')
>>> plt.plot(param_range, test_mean,
...         color='green', linestyle='--',
...         marker='s', markersize=5,
...         label='validation accuracy')
>>> plt.fill_between(param_range,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Parameter C')
>>> plt.ylabel('Accuracy')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

Utilizzando il codice precedente, abbiamo ottenuto la curva di convalida per il parametro `c` (Figura 6.6).

Come nel caso della funzione `learning_curve`, la funzione `validation_curve` utilizza la convalida incrociata k-fold per stimare le prestazioni del modello se utilizza algoritmi di classificazione. All'interno della funzione `validation_curve`, abbiamo specificato il parametro che vogliamo valutare. In questo caso si tratta di `c`, il parametro di

regolarizzazione inversa del classificatore `LogisticRegression`, che abbiamo scritto come `'clf__C'`, per accedere all'oggetto `LogisticRegression` all'interno della pipeline scikit-learn per un determinato intervallo di valori, che abbiamo impostato tramite il parametro `param_range`. In modo simile all'esempio della curva di apprendimento del paragrafo precedente, abbiamo tracciato l'accuratezza media di addestramento e di convalida incrociata e le corrispondenti deviazioni standard.

Sebbene le variazioni di accuratezza per valori variabili di `C` siano limitate, possiamo vedere che il modello soffre di un leggero underfit dei dati quando incrementiamo l'intensità della regolarizzazione (valori piccoli di `C`). Al contrario, per valori di `C` più grandi, riduciamo l'intensità della regolarizzazione e pertanto il modello tende a un leggero overfit sui dati. In questo caso, il valore ottimale sembra centrarsi attorno a `C=0.1`.

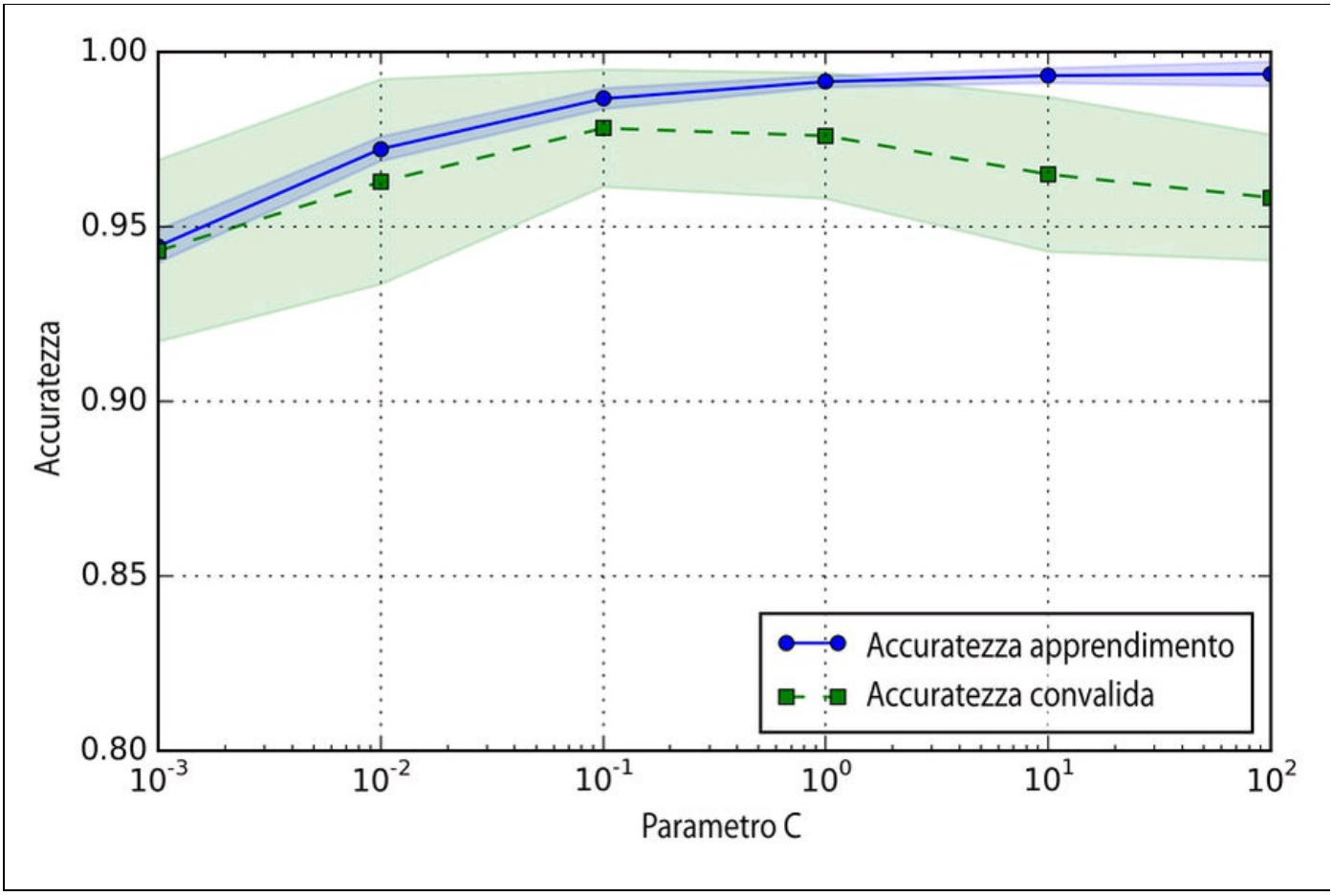


Figura 6.6

Ottimizzazione dei modelli di machine learning tramite ricerca a griglia

Nel campo del machine learning, abbiamo due tipi di parametri: quelli che vengono appresi dai dati di addestramento, per esempio i pesi della regressione logistica, e i parametri di un algoritmo di apprendimento, che vengono ottimizzati separatamente. Questi ultimi sono i parametri di ottimizzazione, o iperparametri, di un modello; per esempio si tratta del parametro di *regolarizzazione* nella regressione logistica o del parametro di *profondità* di un albero decisionale.

Nel paragrafo precedente, abbiamo utilizzato le curve di convalida per migliorare le prestazioni di un modello ottimizzando i suoi iperparametri. In questo paragrafo esamineremo una tecnica potente di ottimizzazione degli iperparametri chiamata *ricerca a griglia*, che può aiutare a migliorare ulteriormente le prestazioni di un modello, ricercando la combinazione *ottimale* dei valori degli iperparametri.

Ottimizzazione degli iperparametri tramite la ricerca a griglia

L'approccio della ricerca a griglia è piuttosto semplice: si tratta di una ricerca esaustiva a forza bruta, nella quale specifichiamo un elenco di valori per i vari iperparametri e il computer valuta le prestazioni del modello per ogni combinazione, fino a ottenere un set ottimale:

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.svm import SVC
>>> pipe_svc = Pipeline([('scl', StandardScaler()),
...                      ('clf', SVC(random_state=1))])
>>> param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{'clf__C': param_range,
...                'clf__kernel': ['linear']},
...               {'clf__C': param_range,
...                'clf__gamma': param_range,
...                'clf__kernel': ['rbf']}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=10,
...                   n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.978021978022
>>> print(gs.best_params_)
{'clf__C': 0.1, 'clf__kernel': 'linear'}
```

Utilizzando il codice precedente, abbiamo inizializzato un oggetto `GridSearchCV` dal modulo `sklearn.grid_search` per addestrare e ottimizzare una pipeline *SVM* (support vector machine). Abbiamo impostato il parametro `param_grid` di `GridSearchCV` con un elenco di

dizionari per specificare i parametri che vorremmo ottimizzare. Per una macchina SVM lineare, abbiamo valutato solo il parametro di regolarizzazione inversa `C`; per una macchina SVM kernel RBF abbiamo ottimizzato i parametri `C` e `gamma`. Notare che il parametro `gamma` è specifico di SVM kernel. Dopo aver utilizzato i dati di addestramento per svolgere la ricerca a griglia, abbiamo ottenuto il punteggio del modello che si comporta meglio (tramite l'attributo `best_score_`) e abbiamo ricercato i set di parametri (cui abbiamo accesso tramite l'attributo `best_params_`). In questo particolare caso, il modello SVM lineare con `'clf__C'= 0.1'` ha fornito la migliore accuratezza di convalida incrociata k-fold: il 97. 8%.

Infine, utilizzeremo il test indipendente per stimare le prestazioni del migliore modello selezionato, che è disponibile tramite l'attributo `best_estimator_` dell'oggetto `GridSearchCV`:

```
>>> clf = gs.best_estimator_
>>> clf.fit(X_train, y_train)
>>> print('Test accuracy: %.3f' % clf.score(X_test, y_test))
Test accuracy: 0.965
```

NOTA

Sebbene la ricerca a griglia sia un approccio potente per la ricerca del set ottimale dei parametri, la valutazione di tutte le possibili combinazioni dei parametri può essere molto costosa dal punto di vista computazionale. Un approccio alternativo al campionamento delle varie combinazioni di parametri utilizzando scikit-learn è la ricerca casuale. Utilizzando la classe `RandomizedSearchCV` di scikit-learn, possiamo estrarre combinazioni parametri casuali dalla distribuzione di campionamento sulla base di un determinato budget. Ulteriori informazioni esempi d'uso si trovano in http://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization.

Selezione dell'algoritmo con convalida incrociata nidificata

L'utilizzo della convalida incrociata k-fold in combinazione con la ricerca a griglia è un approccio utile per l'ottimizzazione delle prestazioni di un modello di machine learning variando i valori dei suoi iperparametri, come abbiamo visto nel paragrafo precedente. Ma se vogliamo selezionare fra vari algoritmi di machine learning, un altro approccio consigliato è la convalida incrociata nidificata e, in un ottimo studio sul bias nella stima degli errori, Varma e Simon hanno concluso che il vero errore sulla stima non è affetto dalla scelta del set di test, qualora venga usata la convalida incrociata nidificata (S. Varma e R. Simon, *Bias in Error Estimation When Using Cross-validation for Model Selection*, in "BMC bioinformatics", 7(1):91, 2006).

Nella convalida incrociata nidificata, abbiamo un ciclo di convalida incrociata k-fold esterno per suddividere i dati fra le parti di addestramento e di test, più un ciclo interno che viene utilizzato per selezionare il modello utilizzando la convalida incrociata k-fold sulla parte di addestramento. Dopo la selezione del modello, la parte di test viene utilizzata per valutare le prestazioni del modello. La Figura 6.7 spiega il concetto della convalida incrociata nidificata con cinque parti esterne e due parti interne, il che può essere utile per grossi dataset nei quali le prestazioni computazionali sono importanti. Questo specifico tipo di convalida incrociata nidificata è chiamata anche *convalida incrociata 5x2*.

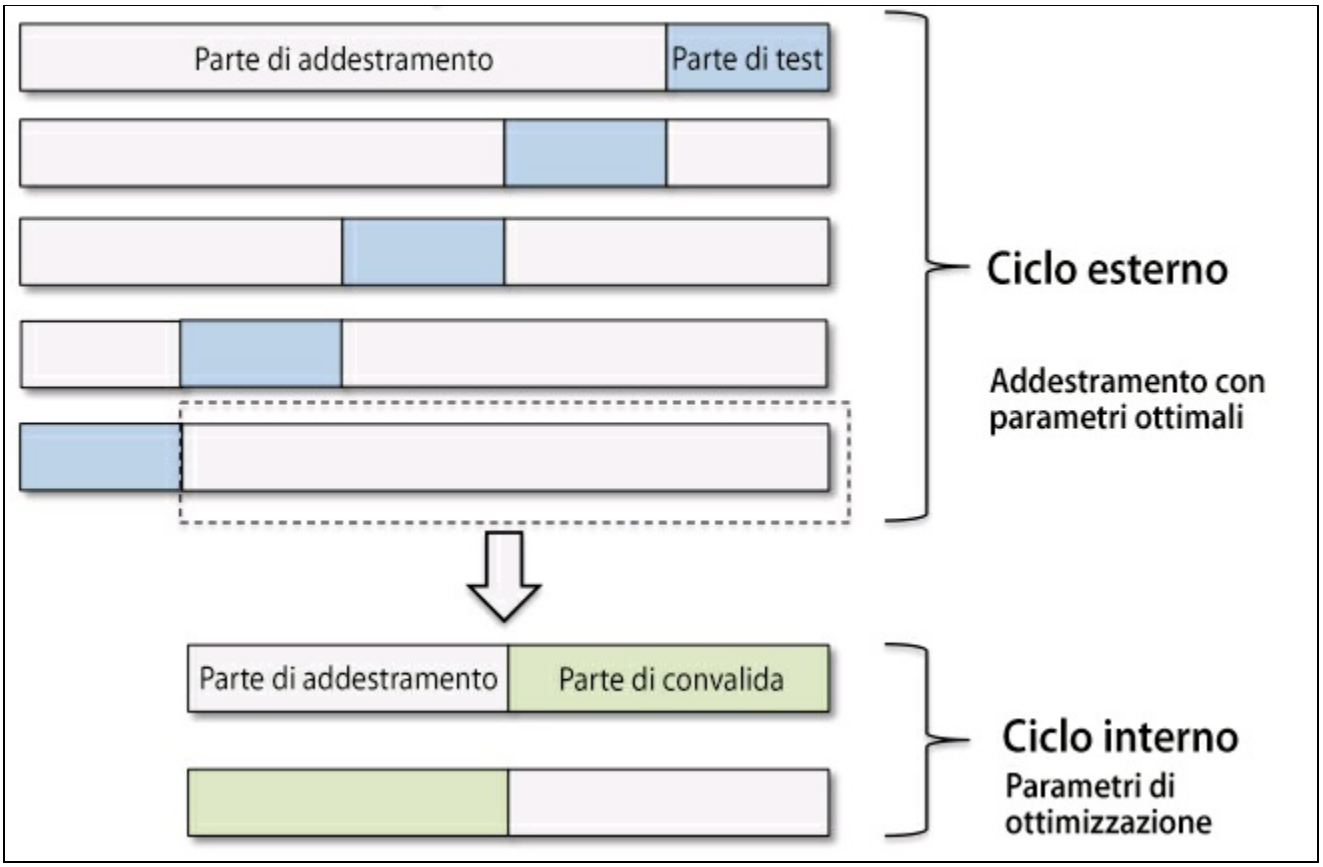


Figura 6.7

In scikit-learn, possiamo eseguire la convalida incrociata nidificata nel seguente modo:

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=2,
...                   n_jobs=-1)
>>> scores = cross_val_score(gs, X_train, y_train, scoring='accuracy', cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...             np.mean(scores), np.std(scores)))
CV accuracy: 0.965 +/- 0.025
```

L'accuratezza media della convalida incrociata che otteniamo ci fornisce una buona stima di cosa possiamo aspettarci se ottimizziamo gli iperparametri di un

modello e poi lo utilizziamo su dati che non ha mai visto prima. Per esempio, possiamo utilizzare l'approccio a convalida incrociata nidificata per confrontare un modello SVM con un semplice classificatore ad albero decisionale; per semplicità, utilizzeremo solo il suo parametro `depth`:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...      estimator=DecisionTreeClassifier(random_state=0),
...      param_grid=[
...          {'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
...      scoring='accuracy',
...      cv=5)
>>> scores = cross_val_score(gs,
...                          X_train,
...                          y_train,
...                          scoring='accuracy',
...                          cv=2)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...                    np.mean(scores), np.std(scores)))
CV accuracy: 0.921 +/- 0.029
```

Come possiamo vedere, le prestazioni determinate dalla convalida incrociata nidificata sul modello SVM (97.8%) sono notevolmente superiori rispetto alle prestazioni dell'albero decisionale (90.8%). Pertanto, ci aspettiamo che questa sia una scelta migliore per la classificazione dei nuovi dati che provengono dalla stessa popolazione di questo specifico dataset.

Varie metriche di valutazione delle prestazioni

Nei paragrafi e nei capitoli precedenti, abbiamo valutato i modelli utilizzando l'accuratezza del modello, la quale è una metrica utile per quantificare le prestazioni di un modello in generale. Tuttavia, esistono varie altre metriche prestazionali che possono essere utilizzate per misurare la rilevanza di un modello: la *precisione*, *recall* e *F1-score*.

Lettura di una matrice di confusione

Prima di entrare nei dettagli delle varie metriche di valutazione, predisponiamo una cosiddetta matrice di confusione, una matrice che rappresenta le prestazioni di un algoritmo di apprendimento. La matrice di confusione è semplicemente una matrice quadrata che rileva il conteggio dei veri positivi e dei veri negativi, dei falsi positivi e dei falsi negativi nelle previsioni di un classificatore, come illustrato dalla Figura 6.8.

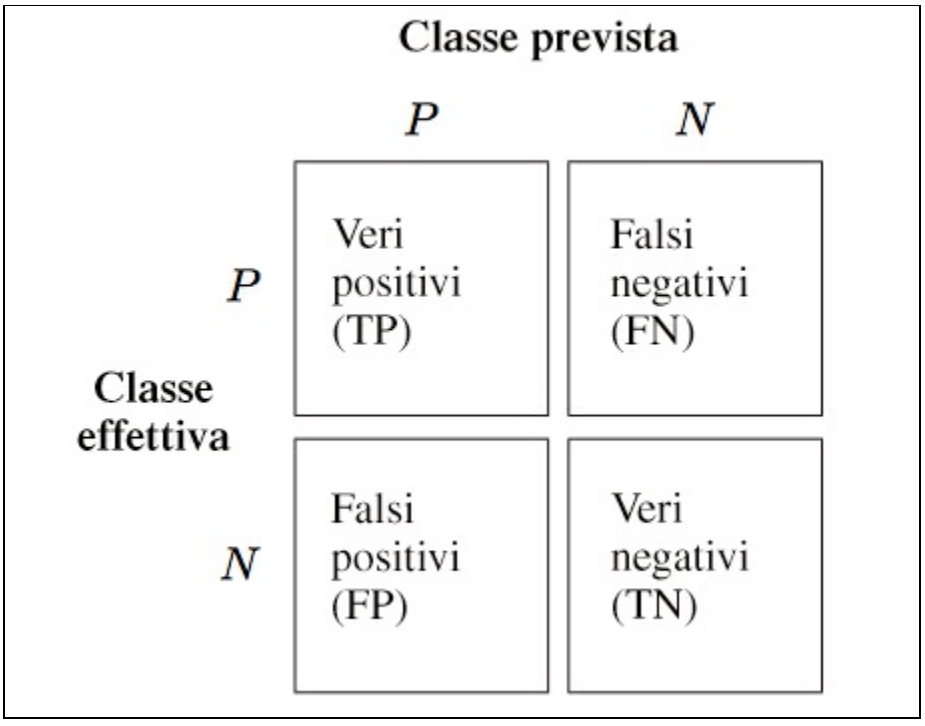


Figura 6.8

Sebbene queste metriche possano essere calcolate con facilità in modo manuale confrontando le etichette delle classi vere e previste, scikit-learn fornisce una comoda funzione `confusion_matrix` che possiamo utilizzare nel seguente modo:

```
>>> from sklearn.metrics import confusion_matrix
>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>>> print(confmat)
[[71  1]
 [ 2 40]]
```

L'array che verrà restituito dopo l'esecuzione del codice precedente ci fornisce informazioni relative ai vari tipi di errori commessi dal classificatore sul dataset di test, che possiamo mappare sulla illustrazione della matrice di confusione della Figura 6.8 precedente utilizzando la funzione matshow di matplotlib:

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i,
...                 s=confmat[i, j],
...                 va='center', ha='center')
>>> plt.xlabel('predicted label')
>>> plt.ylabel('true label')
>>> plt.show()
```

Ora, la matrice di confusione rappresentata nella Figura 6.9 dovrebbe semplificare un po' l'interpretazione dei risultati.

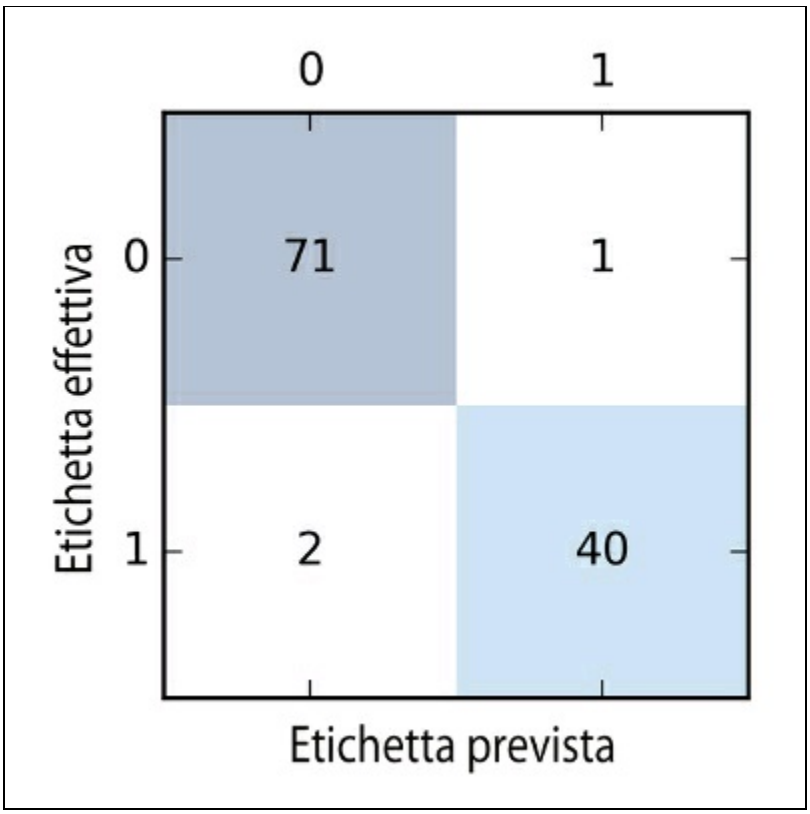


Figura 6.9

Supponendo che la classe 1 (tumore maligno) sia la classe positiva di questo esempio, il nostro modello classifica correttamente 71 dei campioni che appartengono alla classe 0 (veri negativi) e 40 campioni che appartengono alla classe 1 (veri positivi). Tuttavia, il nostro modello ha anche sbagliato a classificare

2 campioni della classe 0, considerandoli della classe 1 (falsi negativi), e ha anche previsto che un campione fosse benigno sebbene si trattasse di un tumore maligno (falso positivo). Nel prossimo paragrafo vedremo come utilizzare questa informazione per calcolare varie metriche d'errore.

Ottimizzazione della precisione e del recall di un modello di classificazione

L'errore (*ERR*) e l'accuratezza (*ACC*) della previsione forniscono entrambe informazioni generali sul numero dei campioni che hanno subito un'errata classificazione. L'errore è dato dalla somma di tutte le false previsioni, divisa per il numero delle previsioni totali; l'accuratezza è data dalla somma delle previsioni corrette divisa per il numero totale delle previsioni:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

L'accuratezza della previsione può essere calcolata direttamente a partire dall'errore:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

Il tasso dei veri positivi (*TPR – true positive rate*) e il tasso dei falsi positivi (*FPR – false positive rate*) sono metriche prestazionali particolarmente utili per studiare i problemi legati a classi fortemente sbilanciate:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

Nella diagnosi dei tumori, per esempio, siamo particolarmente preoccupati del rilevamento dei tumori maligni, in modo da aiutare un paziente e orientarlo verso una cura appropriata. Tuttavia, è anche importante ridurre il numero di tumori benigni che sono stati classificati erroneamente come maligni (falsi positivi) per non allarmare senza motivo un paziente. Al contrario del FPR, il tasso dei veri positivi fornisce informazioni utili relative alla frazione dei campioni positivi (o rilevanti) che sono stati correttamente identificati nel gruppo totale dei positivi (*P*).

La precisione (*PRE*) e la recall (*REC*) sono metriche prestazionali strettamente correlate a questi rapporti di veri positivi e veri negativi; in pratica, recall è sinonimo di tasso dei veri positivi:

$$PRE = \frac{TP}{TP + FP}$$

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

In pratica, spesso viene utilizzata una combinazione di precisione e recall, il cosiddetto punteggio *F1-score*:

$$F1 = 2\frac{PRE \times REC}{PRE + REC}$$

Queste metriche di valutazione sono tutte implementate in scikit-learn e possono essere importate dal modulo `sklearn.metrics`, come illustrato dal seguente frammento di codice:

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics  import recall_score, f1_score
>>> print('Precision: %.3f' % precision_score(
...           y_true=y_test, y_pred=y_pred))
Precision: 0.976
>>> print('Recall: %.3f' % recall_score(
...           y_true=y_test, y_pred=y_pred))
Recall: 0.952
>>> print('F1: %.3f' % f1_score(
...           y_true=y_test, y_pred=y_pred))
F1: 0.964
```

Inoltre, possiamo utilizzare un'altra metrica di valutazione (diversa dall'accuratezza) in `GridSearch` tramite il parametro `scoring`. Un elenco completo dei vari valori che sono accettati dal parametro `scoring` si trova in http://scikit-learn.org/stable/modules/model_evaluation.html.

Ricordate che la classe positiva, in scikit-learn, era la classe etichettata con il numero 1. Se vogliamo specificare un'altra *etichetta positiva*, possiamo costruire un nostro valutatore tramite la funzione `make_scorer`, che possiamo poi fornire direttamente come argomento al parametro `scoring` in `GridSearchCV`:

```
>>> from sklearn.metrics import make_scorer, f1_score
>>> scorer = make_scorer(f1_score, pos_label=0)
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring=scorer,
...                   cv=10)
```

Tracciamento di un ROC

I grafici *ROC* (*Receiver Operator Characteristic*) sono strumenti utili per la selezione dei modelli di classificazione sulla base delle loro prestazioni rispetto ai tassi di falsi positivi e veri positivi, che vengono calcolati spostando la soglia di decisione del classificatore. La diagonale di un grafico ROC può essere interpretata come l'indicazione casuale. I modelli di classificazione che rientrano sotto la diagonale sono considerati peggiori rispetto alla scelta casuale. Un classificatore perfetto si collocherebe all'angolo superiore sinistro del grafico, con un tasso di veri positivi pari a 1 e un tasso di falsi positivi pari a 0. Sulla base della curva ROC, possiamo quindi calcolare la cosiddetta area sotto la curva (*AUC – area under the curve*), per caratterizzare le prestazioni del modello di classificazione.

NOTA

In modo analogo alle curve ROC, possiamo calcolare le *curve precision-recall* per le varie soglie di probabilità di un classificatore. Una funzione per il tracciamento di queste curve precision-recall è implementato anche in scikit-learn e documentato in http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html.

Eseguendo il listato che segue, tracceremo la curva ROC di un classificatore che prova a utilizzare due sole caratteristiche del dataset Breast Cancer Wisconsin per prevedere se un tumore è benigno o maligno. Sebbene utilizziamo la stessa pipeline a regressione logistica che abbiamo definito in precedenza, vogliamo rendere il compito di classificazione più complesso per il classificatore, in modo che la curva ROC risultante divenga visualmente più interessante. Per motivi analoghi, riduciamo anche il numero di parti (`n_folds`) del convalidatore `StratifiedKFold` fino a 3. Il codice è il seguente:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from scipy import interp
>>> pipe_lr = Pipeline([('scl', StandardScaler()),
...                     ('pca', PCA(n_components=2)),
...                     ('clf', LogisticRegression(penalty='l2',
...                                                random_state=0,
...                                                C=100.0))])
>>> X_train2 = X_train[:, [4, 14]]
>>> cv = StratifiedKFold(y_train,
...                      n_folds=3,
...                      random_state=1)
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []
>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(X_train2[train],
                        >>> y_train[train]).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
...                                      probas[:, 1],
...                                      pos_label=1)
...     mean_tpr += interp(mean_fpr, fpr, tpr)
...     mean_tpr[0] = 0.0
...     roc_auc = auc(fpr, tpr)
...     plt.plot(fpr,
```

```
...             tpr,
...             lw=1,
...             label='ROC fold %d (area = %0.2f)'
...                   % (i+1, roc_auc))
>>> plt.plot([0, 1],
...          [0, 1],
...          linestyle='--',
...          color=(0.6, 0.6, 0.6),
...          label='random guessing')
>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...          label='mean ROC (area = %0.2f)' % mean_auc, lw=2)
>>> plt.plot([0, 0, 1],
...          [0, 1, 1],
...          lw=2,
...          linestyle=':',
...          color='black',
...          label='perfect performance')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('false positive rate')
>>> plt.ylabel('true positive rate')
>>> plt.title('Receiver Operator Characteristic')
>>> plt.legend(loc="lower right")
>>> plt.show()
```

Nell'esempio di codice precedente, abbiamo utilizzato la già nota classe `StratifiedKFold` di scikit-learn e abbiamo calcolato le prestazioni ROC del classificatore `LogisticRegression` nella nostra pipeline `pipe_lr` utilizzando la funzione `roc_curve` del modulo `sklearn.metrics` in modo separato per ogni iterazione. Inoltre abbiamo interpolato la curva ROC media dalle tre parti tramite la funzione `interp`, che abbiamo importato da SciPy e abbiamo calcolato l'area sotto la curva tramite la funzione `auc`. La curva ROC risultante indica che un certo grado di varianza fra le varie parti e la media ROC AUC (0.75) si situa fra il punteggio perfetto (1.0) e la scelta casuale (0.5) (Figura 6.10).

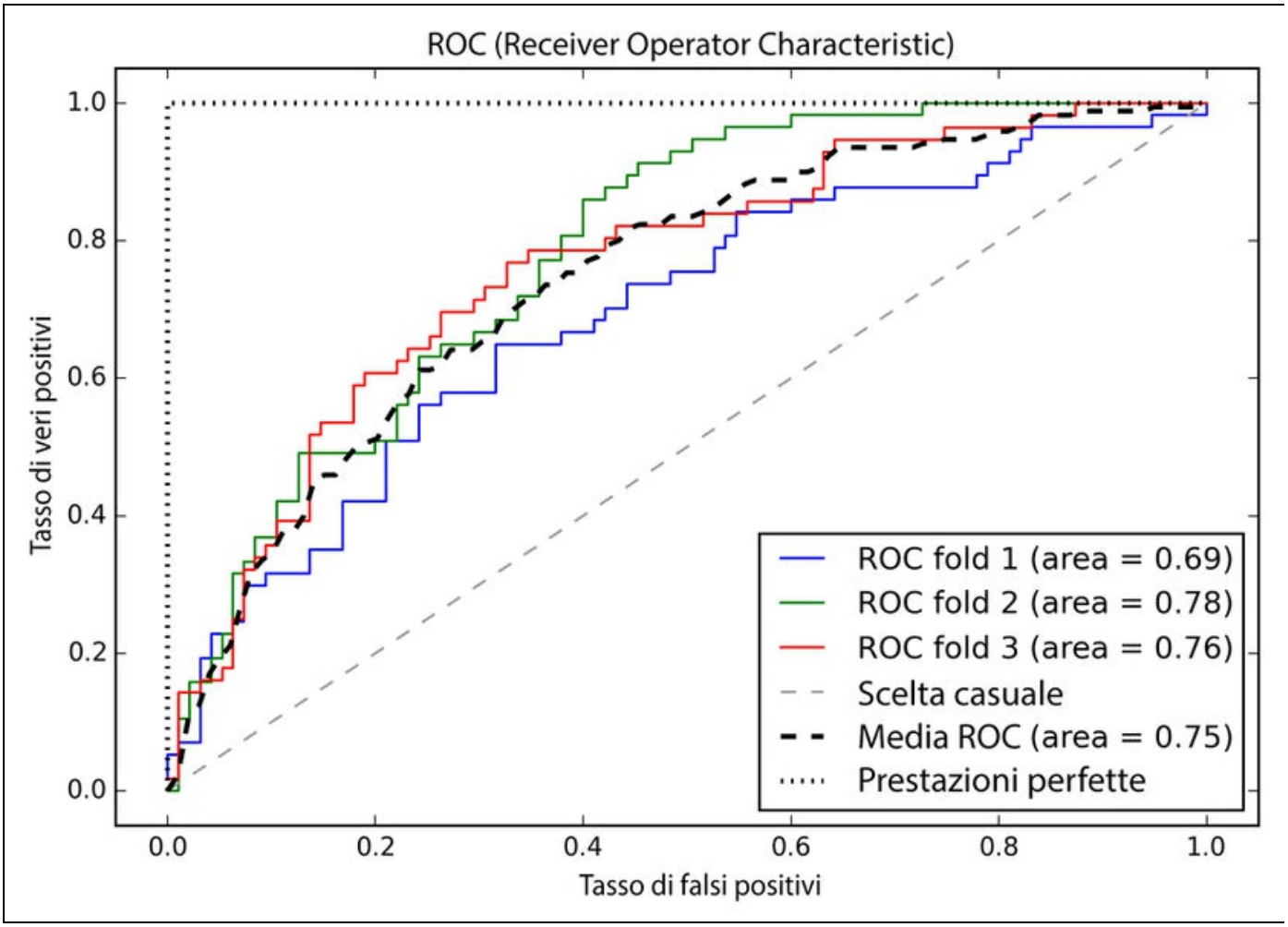


Figura 6.10

Se siamo interessati solo alla valutazione ROC AUC, potremmo anche importare direttamente la funzione `roc_auc_score` dal sottomodulo `sklearn.metrics`. Il codice seguente calcola il punteggio ROC AUC del classificatore su dataset di test indipendenti dopo il loro adattamento sul set di addestramento in due caratteristiche:

```
>>> pipe_lr = pipe_lr.fit(X_train2, y_train)
>>> y_pred2 = pipe_lr.predict(X_test[:, [4, 14]])
>>> from sklearn.metrics import roc_auc_score
>>> from sklearn.metrics import accuracy_score
>>> print('ROC AUC: %.3f' % roc_auc_score(
...       y_true=y_test, y_score=y_pred2))
ROC AUC: 0.662
>>> print('Accuracy: %.3f' % accuracy_score(
...       y_true=y_test, y_pred=y_pred2))
Accuracy: 0.711
```

Le ricerche sulle prestazioni di un classificatore come ROC AUC possono fornire ulteriori informazioni sulle sue prestazioni rispetto a campioni sbilanciati. Tuttavia, mentre il punteggio di accuratezza può essere interpretato come un unico punto discriminante su una curva ROC, A. P. Bradley ha dimostrato che le metriche ROC AUC e di accuratezza in genere concordano l'una con l'altra (A. P. Bradley, *The*

Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms, in "Pattern recognition", 30(7):1145–1159, 1997).

Le metriche di valutazione per la classificazione multiclasse

Le metriche di valutazione di cui abbiamo parlato in questi paragrafi sono specifiche dei sistemi di classificazione binari. Tuttavia, scikit-learn implementa anche metodi di calcolo della media *macro* e *micro* per estendere queste metriche di valutazione ai problemi multiclasse, tramite la classificazione *OvA* (*One vs. All*). La micro-media viene calcolata dai singoli veri positivi, veri negativi, falsi positivi e falsi negativi del sistema. Per esempio, la micro-media del punteggio di precisione in un sistema a k classi può essere calcolata nel seguente modo:

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

La macro-media viene semplicemente calcolata come i punteggi medi dei diversi sistemi:

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

La micro-media è utile se vogliamo pesare ogni istanza o previsione, mentre la macro-media pesa tutte le classi in modo uguale, per valutare le prestazioni globali di un classificatore rispetto alle etichette delle classi più frequenti.

Se utilizziamo le metriche prestazionali binarie per valutare i modelli di classificazione multiclasse in scikit-learn, come impostazione predefinita viene utilizzata una variante normalizzata o pesata della macro-media. La macro-media pesata viene calcolata pesando il punteggio di ciascuna etichetta della classe per il numero di istanze vere nel momento del calcolo della media. La macro-media pesata è utile se abbiamo un problema di scarso equilibrio fra le classi, ovvero quantità differenti di istanze per ciascuna etichetta.

Anche se la macro-media pesata è l'impostazione predefinita per i problemi multiclasse in scikit-learn, possiamo comunque specificare il metodo di calcolo della media tramite il parametro `average` all'interno delle varie funzioni di valutazione che importiamo dal modulo `sklean.metrics`, per esempio le funzioni `precision_score` o `make_scorer`:

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                          pos_label=1,
```

```
...                 greater_is_better=True,
...                 average='micro')
```

Riepilogo

All'inizio di questo capitolo abbiamo visto come concatenare varie tecniche di trasformazione e classificatori in comode pipeline che ci hanno aiutato ad addestrare e a valutare in modo più efficiente i modelli di machine learning. Poi abbiamo utilizzato queste pipeline per svolgere una convalida incrociata k-fold, una delle tecniche fondamentali per la selezione e la valutazione di un modello. Utilizzando la convalida incrociata k-fold, abbiamo tracciato delle curve di apprendimento e di convalida per diagnosticare i problemi più comuni degli algoritmi di apprendimento, in particolare l'overfitting e l'underfitting. Utilizzando la ricerca a griglia abbiamo ulteriormente ottimizzato il nostro modello. Abbiamo concluso il capitolo introducendo la matrice di confusione e varie metriche prestazionali che possono essere utili per ottimizzare ulteriormente le prestazioni di un modello per un determinato problema. Ora abbiamo tutto il necessario in termini di tecniche essenziali per realizzare modelli di machine learning con supervisione in grado di eseguire classificazioni corrette.

Nel prossimo capitolo esamineremo i metodi di assemblaggio, metodi che ci consentono di combinare più modelli e algoritmi di classificazione, con lo scopo di migliorare ulteriormente le prestazioni predittive di un sistema di machine learning.

Combinare più modelli: l'apprendimento d'insieme

Nel capitolo precedente, ci siamo concentrati sulle migliori pratiche di ottimizzazione e di valutazione di vari modelli di classificazione. In questo capitolo ci focalizzeremo su tali tecniche per esplorare altri metodi per la costruzione di un insieme di classificatori a prestazioni predittive che risulta migliore rispetto ai suoi singoli membri. In particolare ci occuperemo dei seguenti argomenti.

- Eseguire previsioni sulla base di un voto a maggioranza.
- Ridurre il problema dell'overfitting estraendo combinazioni casuali del set di addestramento con ripetizione.
- Costruire modelli potenti partendo da *sistemi di apprendimento deboli* che imparano dai loro errori.

Apprendimento d'insieme

L'obiettivo che sta alla base dei *metodi di apprendimento d'insieme* (*ensemble*) consiste nel combinare più classificatori differenti all'interno di un meta-classificatore che offra prestazioni di generalizzazione migliori rispetto a ciascun singolo classificatore. Per esempio, supponendo di aver raccolto le previsioni da dieci esperti, i metodi d'insieme ci consentirebbero di combinare in modo strategico queste dieci previsioni per giungere a una previsione che risulti essere più precisa e solida rispetto alle previsioni di ciascuno dei singoli esperti. Come vedremo più avanti in questo capitolo, vi sono vari approcci per la creazione di un insieme di classificatori. In questo paragrafo presenteremo una descrizione generale su come funzionano gli assiemi e perché viene loro riconosciuta la capacità di fornire buone prestazioni di generalizzazione.

In questo capitolo ci concentreremo sui più noti metodi d'insieme che utilizzano il principio del voto a maggioranza. Il voto a maggioranza significa semplicemente che selezioniamo l'etichetta della classe che è stata prevista dalla maggior parte dei classificatori, ovvero che ha ricevuto più del 50% dei voti. In senso stretto, il termine *voto a maggioranza*, fa riferimento solo a impostazioni relative a classi binarie. Tuttavia, è facile generalizzare il principio del voto a maggioranza anche a impostazioni multiclasse, ottenendo un *voto a pluralità*. Qui selezioniamo l'etichetta della classe che ha ricevuto la maggior parte dei voti (moda). Lo schema rappresentato nella Figura 7.1 illustra il concetto del voto a maggioranza e a pluralità per un insieme di 10 classificatori, dove ciascun simbolo univoco (triangolo, quadrato e cerchio) rappresenta l'etichetta di una classe univoca.

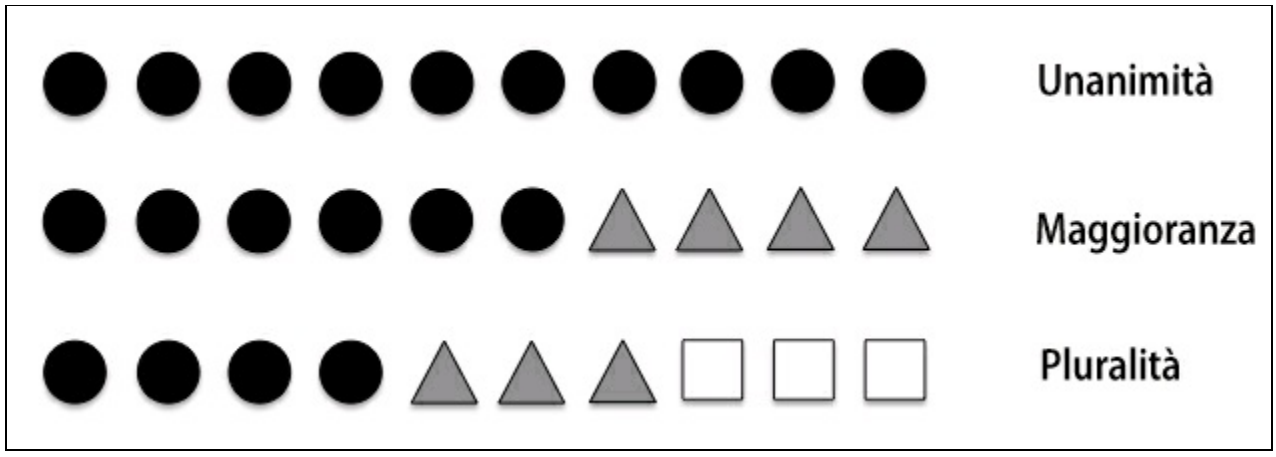


Figura 7.1

Utilizzando l'insieme d'addestramento, iniziamo con l' addestrare *m* classificatori differenti ($C_1,\ldots,C_m$). A seconda della tecnica, l'insieme può essere costruito sulla base di algoritmi di classificazione differenti, per esempio alberi decisionali, ma anche a vettori di supporto, classificatori a regressione logistica e così via. Alternativamente, si può anche utilizzare lo stesso algoritmo di classificazione di base, dotato però di sottoinsiemi differenti di set di addestramento. Un esempio particolarmente evidente di questo approccio potrebbe essere l'algoritmo della foresta casuale, che combina diversi classificatori ad albero decisionale. La Figura 7.2 illustra il concetto di un approccio d'insieme generale che utilizza il voto a maggioranza.

Per prevedere un'etichetta della classe tramite una semplice votazione a maggioranza o pluralità, combiniamo le etichette delle classi previste da ogni singolo classificatore C_j e selezioniamo l'etichetta $\hat{y}$ che ha ricevuto il maggior numero di voti:

$$\hat{y} = mode\left\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\right\}$$

Per esempio, in un compito di classificazione binaria dove $class1 = -1$ e $class2 = +1$, possiamo scrivere la previsione del voto a maggioranza nel seguente modo:

$$C(\boldsymbol{x}) = sign\left[\sum_j^m C_j(\boldsymbol{x})\right] = \begin{cases} 1 \ if \sum_i C_j(\boldsymbol{x}) \geq 0 \\ -1 \ otherwise \end{cases}$$

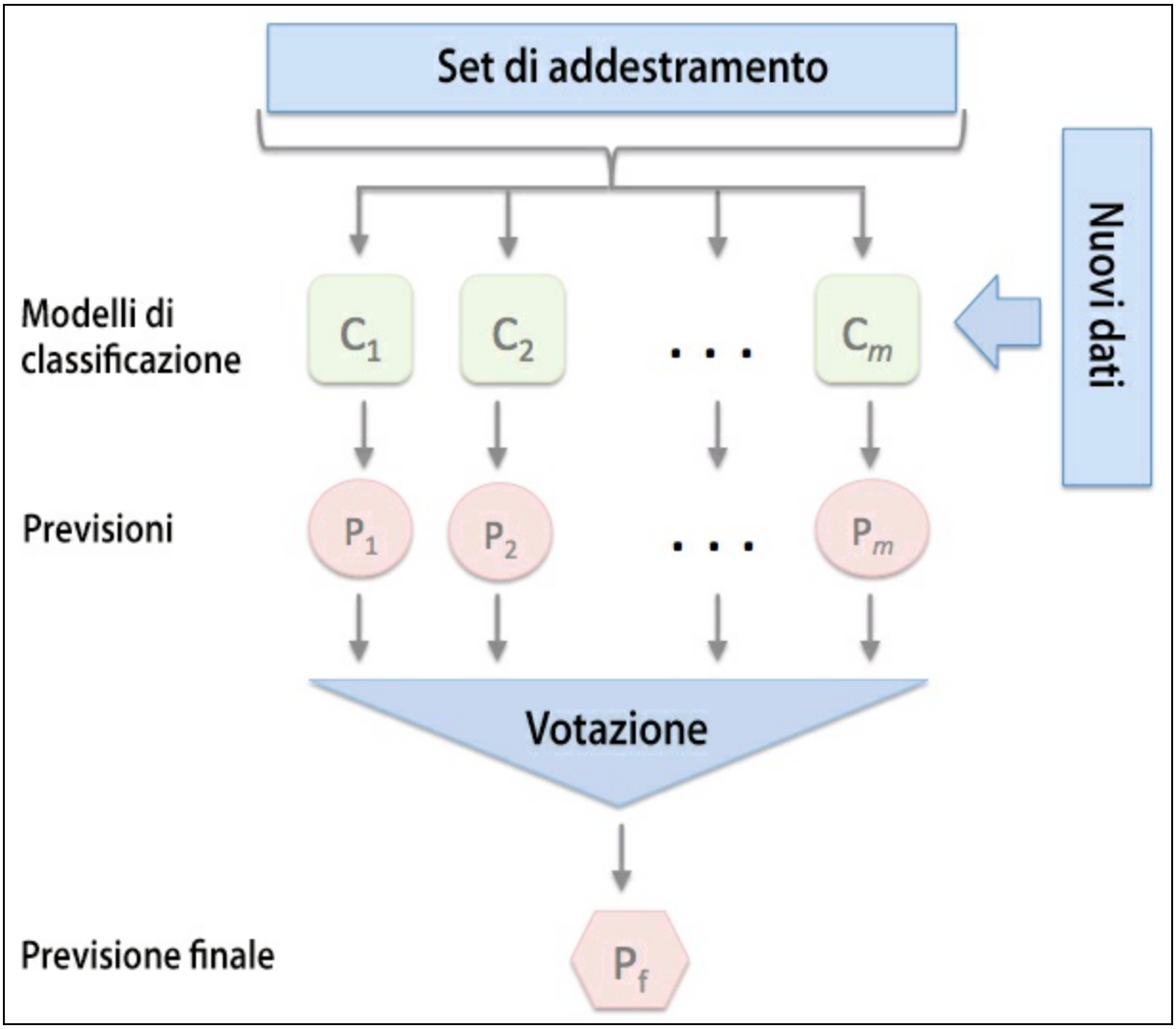


Figura 7.2

Per illustrare perché i metodi d'insieme possono funzionare meglio dei singoli classificatori, applichiamo alcuni semplici concetti combinatoriali. Per il seguente esempio, faremo la supposizione che tutti gli *n* classificatori base per un compito di classificazione binario abbiano un tasso d'errore uguale: ε. Inoltre, supponiamo che i classificatori siano indipendenti e che i tassi d'errore non siano correlati fra loro. Sulla base di queste supposizioni, possiamo semplicemente esprimere la probabilità d'errore di un insieme di classificatori base come la funzione di probabilità di una distribuzione binomiale:

$$P(y \geq k) = \sum_{k}^{n} \left\langle \begin{matrix} n \\ k \end{matrix} \right\rangle \varepsilon^{k} (1-\varepsilon)^{n-k} = \varepsilon_{ensemble}$$

Qui, $\left\langle \begin{matrix} n \\ k \end{matrix} \right\rangle$ è il coefficiente binomiale *n ha scelto k*. In altre parole, calcoliamo la probabilità che la previsione dell'insieme sia errata. Ora esaminiamo un esempio

più concreto di 11 classificatori base ($n = 11$) con un tasso di errore pari a 0.25 ($\varepsilon = 0.25$):

$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1-\varepsilon)^{11-k} = 0.034$$

Come possiamo vedere, il tasso d'errore dell'insieme (0.034) è molto inferiore del tasso d'errore di ogni singolo classificatore (0.25) se tutte le supposizioni sono verificate. Notate che, in questa configurazione semplificata, una suddivisione esatta 50%-50% da parte di un numero pari di classificatori *n* viene trattata come un errore, mentre questo è vero solo nella metà dei casi. Per confrontare questo insieme di classificatori ideale con un classificatore base rispetto a un intervallo di tassi d'errore base differenti, implementiamo la probabilità della funzione di probabilità in Python:

```
>>> from scipy.misc import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = math.ceil(n_classifier / 2.0)
...     probs = [comb(n_classifier, k) *
...              error**k *
...              (1-error)**(n_classifier - k)
...              for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.034327507019042969
```

Dopo aver implementato la funzione `ensemble_error`, possiamo calcolare i tassi d'errore d'insieme per un intervallo di errori base differenti da 0.0 a 1.0 per rappresentare la relazione esistente fra gli errori d'insieme e gli errori base in un grafico a linea:

```
>>> import numpy as np
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...               for error in error_range]
>>> import matplotlib.pyplot as plt
>>> plt.plot(error_range, ens_errors,
...          label='Ensemble error',
...          linewidth=2)
>>> plt.plot(error_range, error_range,
...          linestyle='--', label='Base error',
...          linewidth=2)
>>> plt.xlabel('Base error')
>>> plt.ylabel('Base/Ensemble error')
>>> plt.legend(loc='upper left')
>>> plt.grid()
>>> plt.show()
```

Come possiamo vedere dal grafico rappresentato nella Figura 7.3, la probabilità d'errore di un insieme è sempre migliore rispetto alla probabilità d'errore di un singolo classificatore base, sempre che i classificatori base si comportino meglio della scelta casuale ($\varepsilon < 0.5$). Notate che l'asse *y* misura l'errore base (linea tratteggiata) e l'errore d'insieme (riga linea continua).

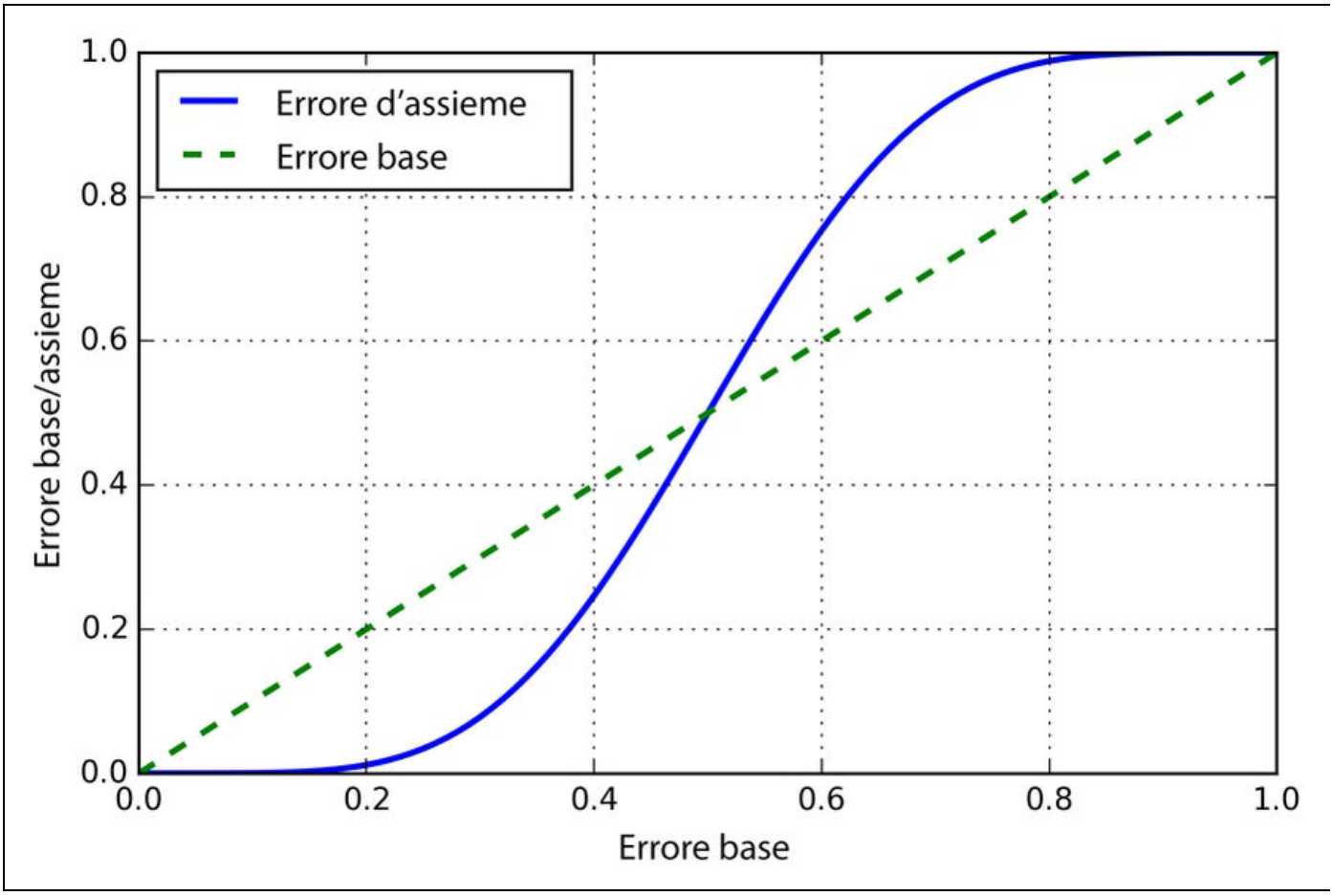

Errore d'assieme
Errore base
Errore base/assieme
Errore base
0.0
0.2
0.4
0.6
0.8
1.0

Figura 7.3

Implementazione di un semplice classificatore con voto a maggioranza

Dopo la breve introduzione all'apprendimento d'insieme del paragrafo precedente, iniziamo con un "esercizio di riscaldamento": implementiamo in Python un semplice classificatore d'insieme per il voto a maggioranza. Sebbene il seguente algoritmo sia generalizzabile a una condizione multiclasse tramite il voto a pluralità, utilizzeremo comunque il termine *voto a maggioranza* per semplicità, in quanto la stessa cosa avviene anche nella letteratura.

L'algoritmo che stiamo per implementare ci consentirà di combinare algoritmi di classificazione differenti associati ai singoli pesi. Il nostro obiettivo è quello di costruire un meta-classificatore più solido, che compensi i punti deboli dei singoli classificatori su uno specifico dataset. In termini più matematici, possiamo scrivere il voto a maggioranza pesato nel seguente modo:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A \left(C_j \left(\boldsymbol{x} \right) = i \right)$$

Qui, w_j è il peso associato al classificatore base C_j, $\hat{y}$ è l'etichetta della classe prevista dall'insieme, χ_A (la lettera greca "*chi*") è la funzione caratteristica $\left[C_j \left(\boldsymbol{x} \right) = i \in A \right]$ e A è l'insieme delle etichette univoche delle classi. Se i pesi sono uguali, possiamo semplificare questa equazione e scriverla nel seguente modo:

$$\hat{y} = mode \left\{ C_1 \left(\boldsymbol{x} \right), C_2 \left(\boldsymbol{x} \right), \ldots, C_m \left(\boldsymbol{x} \right) \right\}$$

Per comprendere meglio il concetto di *peso*, esamineremo un esempio più concreto. Supponiamo di avere un insieme di tre classificatori base C_j (dove $j \in \{0,1\}$) e di voler prevedere l'etichetta della classe di una determinata istanza campione x. Due dei tre classificatori base prevedono che sia l'etichetta della classe 0, mentre C_3 prevede che il campione appartenga alla classe 1. Se pesiamo allo stesso modo le previsioni di ciascuno dei classificatori base, il voto a maggioranza prevederà che il campione appartenga alla classe 0:

$$C_1 \left(x \right) \to 0,\ C_2 \left(x \right) \to 0,\ C_3 \left(x \right) \to 1$$

$$\hat{y} = mode\{0,0,1\} = 0$$

Ora assegniamo un peso pari a 0.6 a C_3 e un peso pari a 0.2 a C_1 e C_2.

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A \left(C_j \left(\boldsymbol{x} \right) = i \right)$$

$$= \arg\max_i \left[0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1 \right] = 1$$

In termini intuitivi, poiché $3 \times 0.2 = 0.6$, possiamo dire che alla previsione restituitaci da C_3 attribuiamo un peso tre volte superiore rispetto alle previsioni effettuate da C_1 o da C_2. Possiamo scrivere questa relazione nel seguente modo:

$$\hat{y} = mode\{0,0,1,1,1\} = 1$$

Per tradurre in codice Python il concetto del voto a maggioranza pesata, possiamo utilizzare le comode funzioni `argmax` e `bincount` di NumPy:

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...          weights=[0.2, 0.2, 0.6]))
1
```

Come abbiamo detto nel Capitolo 3, *I classificatori di machine learning di scikit-learn*, alcuni classificatori di scikit-learn possono anche restituire la probabilità di un'etichetta della classe prevista, utilizzando il metodo `predict_proba`. L'utilizzo delle probabilità della classe prevista al posto delle etichette delle classi per il voto a maggioranza può essere utile se i classificatori del nostro insieme sono ben calibrati. La versione modificata del voto a maggioranza per la previsione delle etichette delle classi a partire dalle probabilità può essere scritta nel seguente modo:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}$$

Qui, p_{ij} è la probabilità prevista del classificatore *jth* per l'etichetta della classe *i*.

Per continuare con l'esempio precedente, supponiamo di avere un problema di classificazione binaria con le etichette della classe $i \in \{0,1\}$ e un insieme di tre classificatori C_j (con $j \in \{1,2,3\}$). Supponiamo che il classificatore C_j restituisca le seguenti probabilità di appartenenza alla classe per un determinato campione $\boldsymbol{x}$:

$$C_1(\boldsymbol{x}) \rightarrow [0.9, 0.1],\ C_2(\boldsymbol{x}) \rightarrow [0.8, 0.2],\ C_3(\boldsymbol{x}) \rightarrow [0.4, 0.6]$$

Quindi calcoliamo le singole probabilità di ogni classe nel seguente modo:

$$p(i_0 \mid \boldsymbol{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1 \mid \boldsymbol{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42$$

$$\hat{y} = \arg\max_i \left[p(i_0 \mid \boldsymbol{x}), p(i_1 \mid \boldsymbol{x}) \right] = 0$$

Per implementare il voto a maggioranza pesato sulla base delle probabilità delle classi, possiamo utilizzare ancora NumPy impiegando numpy.average e np.argmax:

```
>>> ex = np.array([[0.9, 0.1],
...                [0.8, 0.2],
...                [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([ 0.58,  0.42])
>>> np.argmax(p)
0
```

Riepilogando il tutto, implementiamo in Python un classificatore MajorityVoteClassifier:

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.externals import six
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator


class MajorityVoteClassifier(BaseEstimator,
                             ClassifierMixin):
    """ A majority vote ensemble classifier


    Parameters
    ----------
    classifiers : array-like, shape = [n_classifiers]
      Different classifiers for the ensemble
    vote : str, {'classlabel', 'probability'}
      Default: 'classlabel'
      If 'classlabel' the prediction is based on
      the argmax of class labels. Else if
      'probability', the argmax of the sum of
      probabilities is used to predict the class label
      (recommended for calibrated classifiers).
    weights : array-like, shape = [n_classifiers]
      Optional, default: None
      If a list of `int` or `float` values are
      provided, the classifiers are weighted by
      importance; Uses uniform weights if `weights=None`.

    """
    def __init__(self, classifiers,
                 vote='classlabel', weights=None):
        self.classifiers = classifiers
        self.named_classifiers = {key: value for
                                  key, value in
```

```
                    _name_estimators(classifiers)}
    self.vote = vote
    self.weights = weights

  def fit(self, X, y):
    """ Fit classifiers.

    Parameters
    ----------
    X : {array-like, sparse matrix},
      shape = [n_samples, n_features]
      Matrix of training samples.
    y : array-like, shape = [n_samples]
      Vector of target class labels.

    Returns
    -------
    self : object
    """
    # Use LabelEncoder to ensure class labels start
    # with 0, which is important for np.argmax
    # call in self.predict
    self.lablenc_ = LabelEncoder()
    self.lablenc_.fit(y)
    self.classes_ = self.lablenc_.classes_
    self.classifiers_ = []
    for clf in self.classifiers:
      fitted_clf = clone(clf).fit(X,
                  self.lablenc_.transform(y))
      self.classifiers_.append(fitted_clf)
    return self
```

I commenti presenti nel codice dovrebbero aiutare a comprendere le singole parti. Tuttavia, prima di implementare i metodi rimanenti, prendiamoci una piccola pausa per parlare di una parte di codice che può sembrare un po' complessa. Abbiamo utilizzato le classi genitore `BaseEstimator` e `ClassifierMixin` per ottenere alcune caratteristiche di base in modo "gratis", includendo i metodi `get_params` e `set_params` per impostare e restituire i parametri del classificatore e anche il metodo `score` per calcolare l'accuratezza della previsione. Inoltre notate che abbiamo importato `six` per rendere `MajorityVoteClassifier` compatibile con Python 2.7.

Ora aggiungeremo il metodo `predict` per prevedere l'etichetta della classe tramite voto a maggioranza, inizializzando un nuovo oggetto `MajorityVoteClassifier` con `vote='classlabel'`. Alternativamente, potremo inizializzare il classificatore d'insieme con `vote='probability'` per prevedere l'etichetta della classe sulla base delle probabilità di appartenenza alla classe. Inoltre, utilizzeremo anche un metodo `predict_proba` per restituire le probabilità medie, utili per calcolare l'area sotto la curva (*ROC AUC – Characteristic area under the curve*).

```
  def predict(self, X):
    """ Predict class labels for X.

    Parameters
    ----------
    X : {array-like, sparse matrix},
      Shape = [n_samples, n_features]
      Matrix of training samples.

    Returns
    ----------
```

```
        maj_vote : array-like, shape = [n_samples]
            Predicted class labels.

        """
        if self.vote == 'probability':
            maj_vote = np.argmax(self.predict_proba(X),
                                 axis=1)
        else:  # 'classlabel' vote
            #  Collect results from clf.predict calls
            predictions = np.asarray([clf.predict(X)
                                      for clf in
                                      self.classifiers_]).T
            maj_vote = np.apply_along_axis(
                       lambda x:
                       np.argmax(np.bincount(x,
                                 weights=self.weights)),
                       axis=1,
                       arr=predictions)
        maj_vote = self.lablenc_.inverse_transform(maj_vote)
        return maj_vote

    def predict_proba(self, X):
        """ Predict class probabilities for X.

        Parameters
        ----------
        X : {array-like, sparse matrix},
            shape = [n_samples, n_features]
            Training vectors, where n_samples is
            the number of samples and
            n_features is the number of features.

        Returns
        ----------
        avg_proba : array-like,
            shape = [n_samples, n_classes]
            Weighted average probability for
            each class per sample.

        """
        probas = np.asarray([clf.predict_proba(X)
                             for clf in self.classifiers_])
        avg_proba = np.average(probas,
                               axis=0, weights=self.weights)
        return avg_proba

    def get_params(self, deep=True):
        """ Get classifier parameter names for GridSearch"""
        if not deep:
            return super(MajorityVoteClassifier,
                         self).get_params(deep=False)
        else:
            out = self.named_classifiers.copy()
            for name, step in\
                    six.iteritems(self.named_classifiers):
                for key, value in six.iteritems(
                        step.get_params(deep=True)):
                    out['%s__%s' % (name, key)] = value
            return out
```

Inoltre, notate che abbiamo definito la nostra versione modificata dei metodi `get_params` per utilizzare la funzione `_name_estimators` in modo da accedere ai parametri dei singoli classificatori dell'insieme. Inizialmente questo può sembrare un po' complicato, ma tutto avrà perfettamente senso utilizzando la ricerca a griglia per l'ottimizzazione degli iperparametri, come faremo nei prossimi paragrafi.

NOTA

Sebbene la nostra implementazione di `MajorityVoteClassifier` sia molto utile per scopi dimostrativi, abbiamo implementato anche una versione più sofisticata del classificatore con voto a

maggioranza in scikit-learn. Questa diverrà disponibile come sklearn.ensemble.VotingClassifier nella prossima release (versione 0.17).

Combinare algoritmi differenti per la classificazione con voto a maggioranza

Ora è giunto il momento di mostrare in azione il classificatore MajorityVoteClassifier che abbiamo implementato nel paragrafo precedente. Ma prima prepariamo un dataset sul quale eseguire il test. Poiché conosciamo già le tecniche di caricamento di dataset da file, prenderemo la via rapida e caricheremo il dataset *Iris* dal modulo dei dataset di scikit-learn. Inoltre, selezioneremo solo due caratteristiche, *sepal width* e *petal length*, con lo scopo di rendere più complesso il compito di classificazione. Sebbene il nostro classificatore MajorityVoteClassifier sia generalizzabile a problemi multiclasse, classificheremo solo i campioni di fiori di due classi, *Iris-Versicolor* e *Iris-Virginica*, per calcolare l'area *ROC AUC*. Il codice è il seguente:

```
>>> from sklearn import datasets
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

NOTA

Notate che scikit-learn usa il metodo predict_proba (se applicabile) per calcolare la valutazione ROC AUC. Nel Capitolo 3, *I classificatori di machine learning di scikit-learn*, abbiamo visto come le probabilità di una classe vengano calcolate tramite modelli a regressione logistica. Negli alberi decisionali, le probabilità vengono calcolate da un vettore delle frequenze che viene creato per ciascun nodo al momento dell'addestramento. Il vettore raccoglie i valori di frequenza di ciascuna etichetta della classe calcolata dalla distribuzione delle etichette delle classi in quel nodo. Poi le frequenze vengono normalizzate in modo che la somma sia 1. Analogamente, le etichette delle classi dei k-vicini più prossimi vengono aggregate per restituire le frequenze normalizzate delle etichette delle classi tramite l'algoritmo dei k-vicini più prossimi. Sebbene le probabilità normalizzate restituite sia dal classificatore ad albero decisionale sia dal classificatore a k-vicini più prossimi possono sembrare simili alle probabilità ottenute da un modello a regressione logistica, dobbiamo considerare che esse in realtà non sono derivate dalle funzioni di probabilità.

Poi suddividiamo i campioni di Iris in un 50% di addestramento e 50% di test:

```
>>> X_train, X_test, y_train, y_test =\
...     train_test_split(X, y,
...                      test_size=0.5,
...                      random_state=1)
```

Utilizzando il dataset di addestramento, addestreremo ora tre diversi classificatori (un classificatore a regressione logistica, un classificatore ad albero decisionale e un

classificatore a k-vicini più prossimi) e osserveremo le loro prestazioni tramite una convalida incrociata a 10 parti sul dataset di addestramento, prima di combinare il tutto in un classificatore d'insieme:

```
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           random_state=0)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                               criterion='entropy',
...                               random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf1]])
>>> pipe3 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf3]])
>>> clf_labels = ['Logistic Regression', 'Decision Tree', 'KNN']
>>> print('10-fold cross validation:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
>>>                              X=X_train,
>>>                              y=y_train,
>>>                              cv=10,
>>>                              scoring='roc_auc')
>>>     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...           % (scores.mean(), scores.std(), label))
```

L'output che otteniamo, come si può vedere dal seguente frammento di codice, mostra che le prestazioni predittive dei singoli classificatori sono all'incirca le stesse:

```
10-fold cross validation:
ROC AUC: 0.92 (+/- 0.20) [Logistic Regression]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
```

Potreste chiedervi perché abbiamo addestrato i classificatori a regressione logistica e a k-vicini più prossimi nell'ambito di una *pipeline*. Il motivo è che, come abbiamo detto nel Capitolo 3, *I classificatori di machine learning di scikit-learn*, entrambi gli algoritmi (utilizzando la metrica della distanza euclidea) non sono invarianti rispetto alla scala, al contrario degli alberi decisionali. Sebbene le caratteristiche dei fiori non diano problemi in quanto sono tutte misurate sulla stessa scala (cm) è buona abitudine operare con caratteristiche standardizzate.

Ora passiamo alla parte più interessante e combiniamo i singoli classificatori per il voto a maggioranza nel nostro MajorityVoteClassifier:

```
>>> mv_clf = MajorityVoteClassifier(
...               classifiers=[pipe1, clf2, pipe3])
>>> clf_labels += ['Majority Voting']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
...                              cv=10,
```

```
...                          scoring='roc_auc')
...     print("Accuracy: %0.2f (+/- %0.2f) [%s]"
...               % (scores.mean(), scores.std(), label))
ROC AUC: 0.92 (+/- 0.20) [Logistic Regression]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
ROC AUC: 0.97 (+/- 0.10) [Majority Voting]
```

Come possiamo vedere, le prestazioni di MajorityVotingClassifier sono sensibilmente migliorate rispetto ai singoli classificatori nella valutazione della convalida incrociata a 10 parti.

Valutazione e ottimizzazione del classificatore d'insieme

In questo paragrafo ci occuperemo di calcolare le curve ROC del set di test per controllare se il classificatore MajorityVoteClassifier generalizza bene su dati mai visti prima. Dovremmo ricordare che il set di test non deve essere utilizzato per la selezione del modello; il suo unico scopo è quello di rilevare una stima non distorta delle prestazioni di generalizzazione di un sistema classificatore. Il codice è il seguente:

```
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyles = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...         in zip(all_clf, clf_labels, colors, linestyles):
...     # assuming the label of the positive class is 1
...     y_pred = clf.fit(X_train,
...                      y_train).predict_proba(X_test)[:, 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                      y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...              color=clr,
...              linestyle=ls,
...              label='%s (auc = %0.2f)' % (label, roc_auc))
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...          linestyle='--',
...          color='gray',
...          linewidth=2)
>>> plt.xlim([-0.1, 1.1])
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid()
>>> plt.xlabel('False Positive Rate')
>>> plt.ylabel('True Positive Rate')
>>> plt.show()
```

Come possiamo vedere nel grafico risultante (Figura 7.4), il classificatore d'insieme si comporta bene anche sul set di test (*ROC AUC = 0.95*), mentre il classificatore a k-vicini più prossimi sembra subire un problema di overfitting dei dati di addestramento (addestramento *ROC AUC = 0.93*, test *ROC AUC = 0.86*).

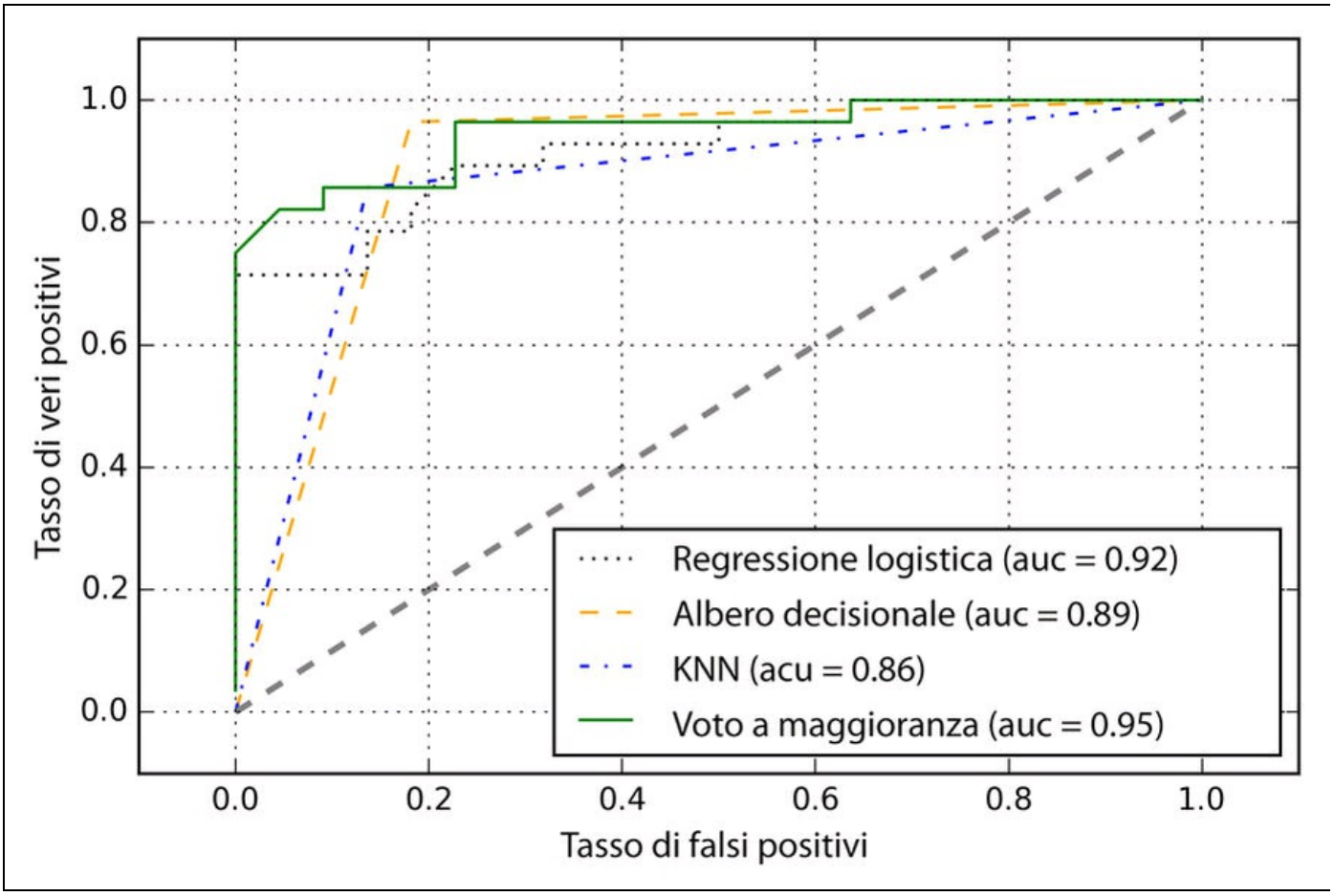


Figura 7.4

Poiché abbiamo solo selezionato due caratteristiche per questo esempio di classificazione, sarebbe interessante vedere qual è l'aspetto effettivo della ricerca a regione decisionale del classificatore d'insieme. Sebbene non sia necessario standardizzare le caratteristiche di addestramento prima dell'adattamento del modello, poiché se ne occuperanno automaticamente le nostre pipeline per la regressione logistica e i k-vicini più prossimi, standardizzeremo il set addestramento in modo che le regioni decisionali dell'albero decisionale siano nella stessa scala, e ciò per puro scopo di rappresentazione grafica. Il codice è il seguente.

```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>> y_max = X_train_std[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                         all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
```

```
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
...                                   X_train_std[y_train==0, 1],
...                                   c='blue',
...                                   marker='^',
...                                   s=50)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...                                   X_train_std[y_train==1, 1],
...                                   c='red',
...                                   marker='o',
...                                   s=50)
...     axarr[idx[0], idx[1]].set_title(tt)
>>> plt.text(-3.5, -4.5,
...          s='Sepal width [standardized]',
...          ha='center', va='center', fontsize=12)
>>> plt.text(-10.5, 4.5,
...          s='Petal length [standardized]',
...          ha='center', va='center',
...          fontsize=12, rotation=90)
>>> plt.show()
```

È interessante notare, ma anche prevedibile, che le regioni decisionali del classificatore d'insieme sembrano essere un ibrido delle regioni decisionali dei singoli classificatori. A una prima occhiata, il confine decisionale del voto a maggioranza assomiglia molto al confine decisionale del classificatore a k-vicini più prossimi. Tuttavia, possiamo vedere che è ortogonale all'asse *y* per $sepal\ width \geq 1$, come l'albero decisionale (Figura 7.5).

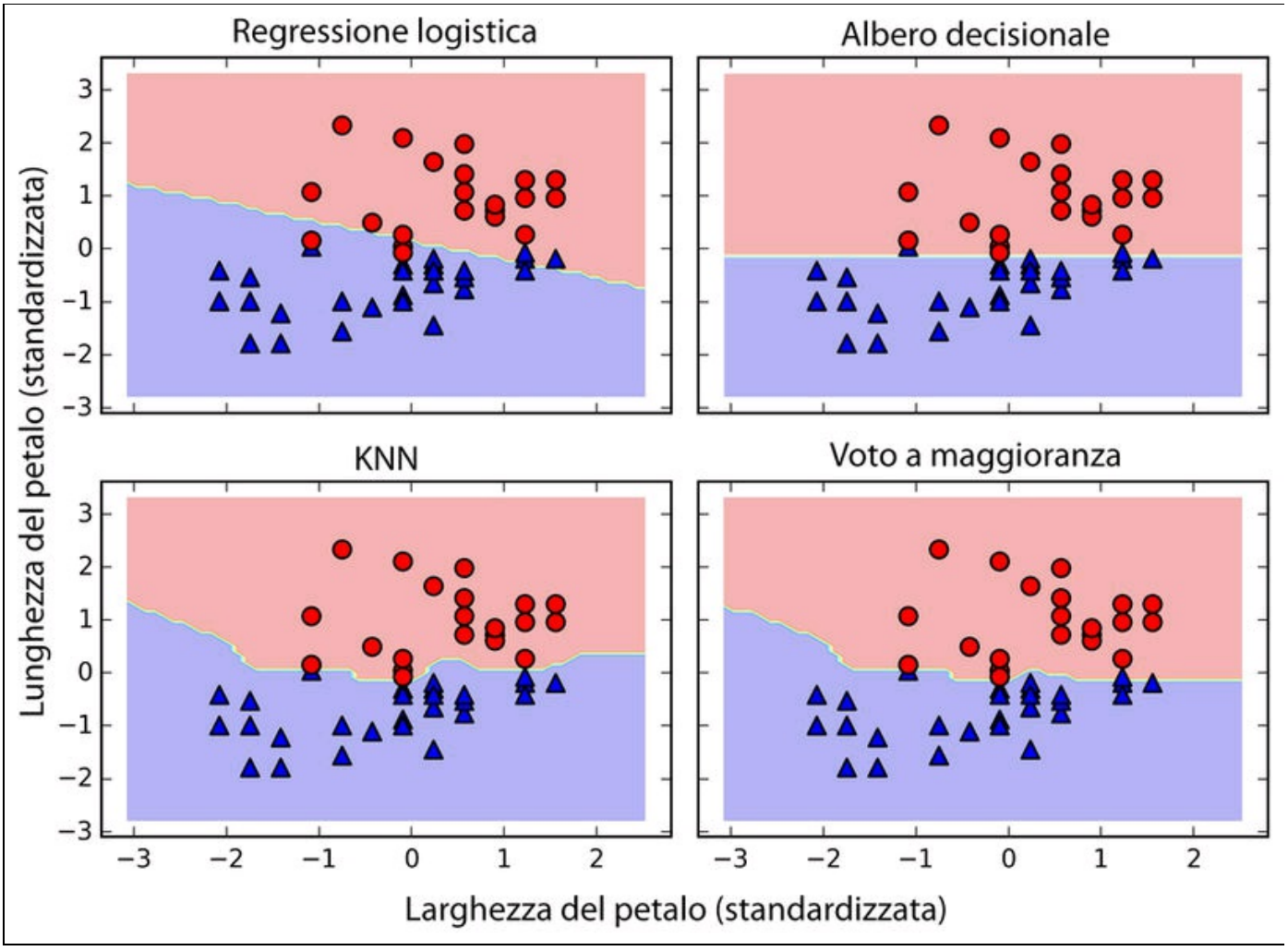


Figura 7.5

Prima di imparare a ottimizzare i singoli parametri del classificatore d'insieme, richiamiamo il metodo `get_params` per avere un'idea di come possiamo accedere ai singoli parametri che si trovano all'interno dell'oggetto `GridSearch`:

```
>>> mv_clf.get_params()
{'decisiontreeclassifier': DecisionTreeClassifier(class_weight=None, criterion='entropy',
      max_depth=1, max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
      min_samples_split=2, min_weight_fraction_leaf=0.0, random_state=0, splitter='best'),
   'decisiontreeclassifier__class_weight': None,
   'decisiontreeclassifier__criterion': 'entropy',
    [...]
   'decisiontreeclassifier__random_state': 0,
   'decisiontreeclassifier__splitter': 'best',
   'pipeline-1': Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
      with_std=True)), ('clf', LogisticRegression(C=0.001, class_weight=None, dual=False,
      fit_intercept=True, intercept_scaling=1, max_iter=100, multi_class='ovr',
      penalty='l2', random_state=0, solver='liblinear', tol=0.0001, verbose=0))]),
   'pipeline-1__clf': LogisticRegression(C=0.001, class_weight=None, dual=False,
      fit_intercept=True, intercept_scaling=1, max_iter=100, multi_class='ovr',
      penalty='l2', random_state=0, solver='liblinear', tol=0.0001, verbose=0),
   'pipeline-1__clf__C': 0.001,
   'pipeline-1__clf__class_weight': None,
   'pipeline-1__clf__dual': False,
    [...]
   'pipeline-1__sc__with_std': True,
   'pipeline-2': Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
      with_std=True)), ('clf', KNeighborsClassifier(algorithm='auto', leaf_size=30,
      metric='minkowski', metric_params=None, n_neighbors=1, p=2, weights='uniform'))]),
   'pipeline-2__clf': KNeighborsClassifier(algorithm='auto', leaf_size=30,
      metric='minkowski', metric_params=None, n_neighbors=1, p=2, weights='uniform'),
   'pipeline-2__clf__algorithm': 'auto',
    [...]
   'pipeline-2__sc__with_std': True}
```

Sulla base dei valori restituiti dal metodo `get_params`, ora sappiamo come accedere agli attributi dei singoli classificatori. Ottimizzeremo quindi il parametro di regolarizzazione inversa `C` del classificatore a regressione logistica e la profondità dell'albero decisionale tramite una ricerca a griglia per puri scopi dimostrativi. Il codice è il seguente:

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {'decisiontreeclassifier__max_depth': [1, 2],
...          'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                     param_grid=params,
...                     cv=10,
...                     scoring='roc_auc')
>>> grid.fit(X_train, y_train)
```

Al termine della ricerca a griglia, possiamo stampare le varie combinazioni di valori degli iperparametri e le valutazioni medie ROC AUC calcolate tramite una convalida incrociata a 10 parti. Il codice è il seguente:

```
>>> for params, mean_score, scores in grid.grid_scores_:
...     print("%0.3f+/-%0.2f %r"
...           % (mean_score, scores.std() / 2, params))
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_depth': 1}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__max_depth': 2}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_depth': 2}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__max_depth': 2}
>>> print('Best parameters: %s' % grid.best_params_)
```

```
Best parameters: {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__max_depth': 1}
>>> print('Accuracy: %.2f' % grid.best_score_)
Accuracy: 1.00
```

Come possiamo vedere, otteniamo i migliori risultati nella convalida incrociata quando scegliamo un'intensità di regolarizzazione più bassa ($C = 100.0$) mentre la profondità dell'albero non sembra influenzare affatto le prestazioni, suggerendo che il primo valore sia sufficiente per separare i dati. Per ricordarci che non è una buona abitudine utilizzare il dataset di test più di una volta per la valutazione di un modello, in questo paragrafo non eseguiremo alcuna stima delle prestazioni di generalizzazione degli iperparametri ottimizzati. Esamineremo invece un approccio alternativo all'apprendimento d'insieme: il *bagging*.

NOTA

L'approccio con voto a maggioranza che abbiamo implementato in questo paragrafo è chiamato anche *stacking*. Tuttavia, l'algoritmo stacking viene tipicamente utilizzato soprattutto in combinazione con il modello a regressione logistica che prevede l'etichetta finale della classe utilizzando come input tutte le previsioni dei singoli classificatori dell'insieme, come descritto più in dettaglio da David H. Wolpert in D. H. Wolpert, *Stacked generalization*, in "Neural networks", 5(2):241–259, 1992.

Bagging: costruire un insieme di classificatori da campioni di bootstrap

Il *bagging* è una tecnica di apprendimento d'insieme strettamente correlata con il `MajorityVoteClassifier` che abbiamo implementato nel paragrafo precedente, come illustrato dalla Figura 7.6.

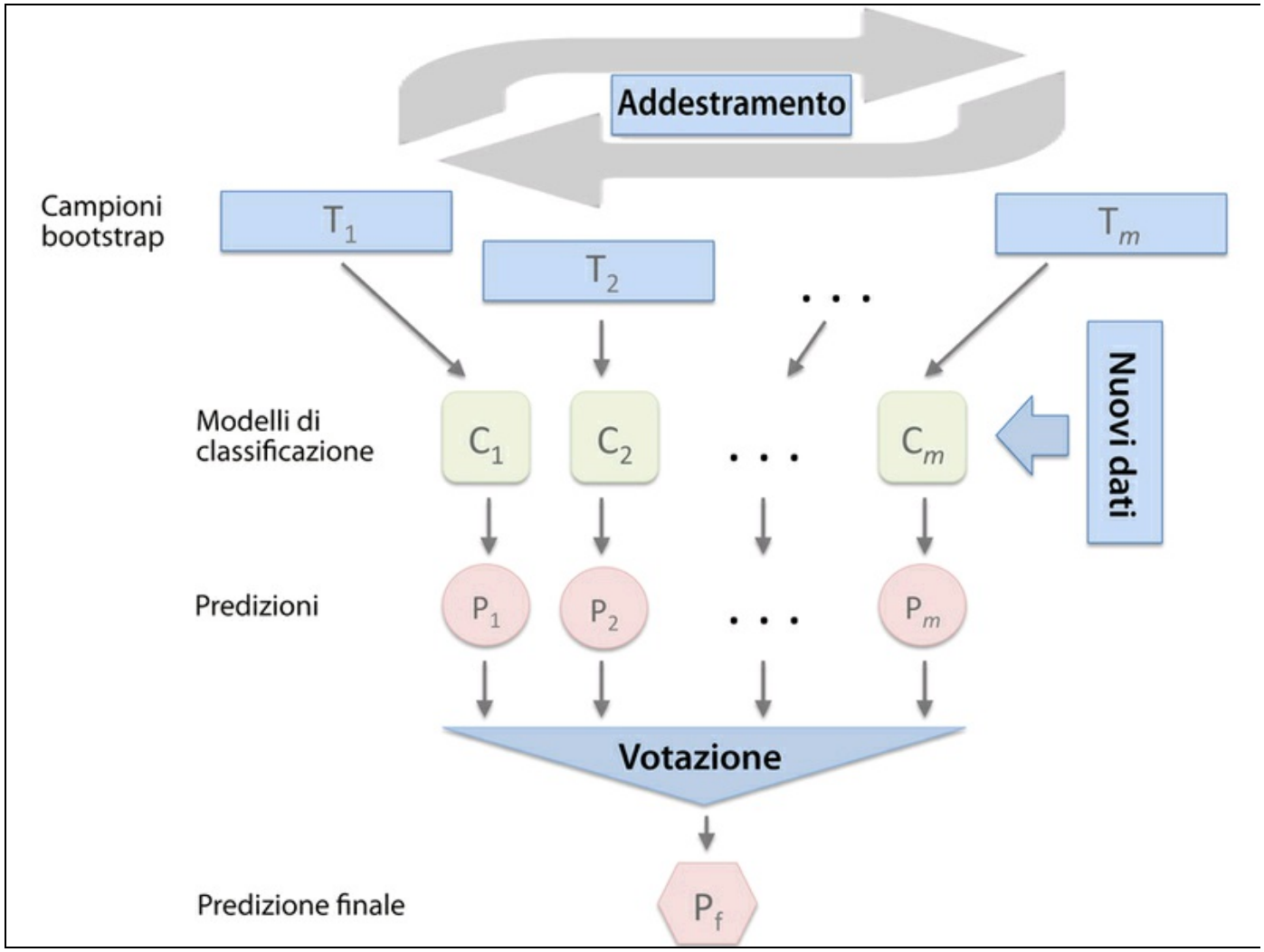


Figura 7.6

Tuttavia, invece di utilizzare lo stesso set di addestramento per adattare i singoli classificatori dell'insieme, estraiamo dal set di addestramento iniziale dei campioni di bootstrap (campioni casuali con reinserimento), motivo per cui la tecnica di bagging è chiamata anche *aggregazione a bootstrap*. Per fornire un esempio più concreto di come funzioni la tecnica di bootstrap, consideriamo l'esempio rappresentato nella Figura 7.7. Qui abbiamo sette diverse istanze di addestramento (denotate dagli indici compresi fra 1 e 7) che sono campionati in modo casuale con

reinserimento a ciascuna fase del bagging. Ogni campione di bootstrap viene poi utilizzato per adattare un classificatore C_j, che normalmente è un albero decisionale non potato.

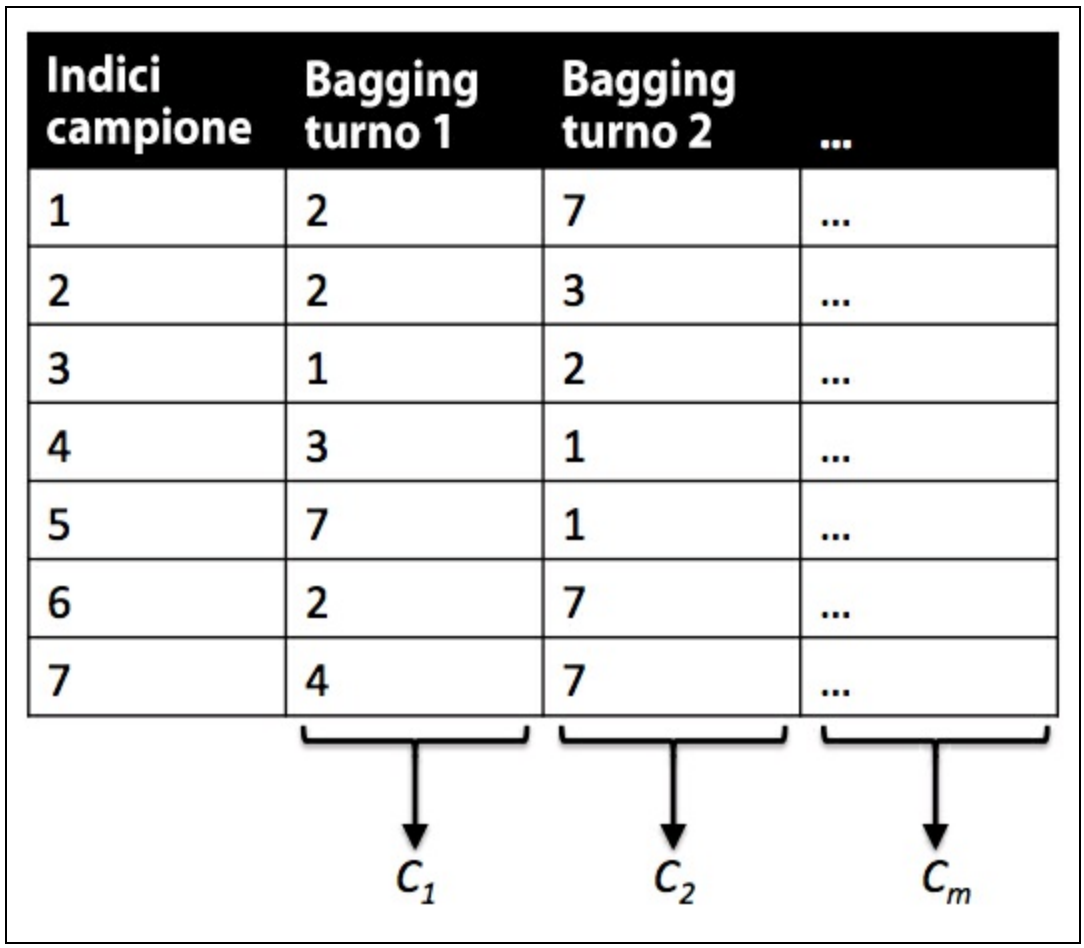


Figura 7.7

Il bagging è anche in correlazione con il classificatore a foresta casuale che abbiamo introdotto nel Capitolo 3, *I classificatori di machine learning di scikit-learn*. In realtà, le foreste casuali possono essere considerate come un caso speciale del bagging, dove utilizziamo anche sottoinsiemi di caratteristiche casuali per adattare i singoli alberi decisionali. La tecnica di bagging è stata proposta per la prima volta da Leo Breiman in un report tecnico nel 1994; egli ha dimostrato che la tecnica bagging può migliorare l'accuratezza dei modelli instabili e ridurre il grado di overfitting. A tale proposito si consiglia la lettura di L. Breiman, *Bagging Predictors*, in "Machine Learning", 24(2):123–140, 1996, liberamente disponibile online.

Per vedere in azione la tecnica di bagging, creiamo un problema di classificazione più complesso, utilizzando il dataset *Wine* che abbiamo introdotto nel Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione*. Qui,

considereremo solo le classi Wine 2 e 3 e selezioneremo due caratteristiche: *Alcohol* e *Hue*.

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash',
...                    'Magnesium', 'Total phenols',
...                    'Flavanoids', 'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>> df_wine = df_wine[df_wine['Class label'] != 1]
>>> y = df_wine['Class label'].values
>>> X = df_wine[['Alcohol', 'Hue']].values
```

Poi modifichiamo le etichette delle classi in formato binario e suddividiamo il dataset in un 60% di addestramento e in un 40% di test:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.cross_validation import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test =\
...          train_test_split(X, y,
...                           test_size=0.40,
...                           random_state=1)
```

L'algoritmo `BaggingClassifier` è già implementato in scikit-learn, e possiamo importarlo dal modulo `ensemble`. Utilizzeremo quale classificatore di base un albero decisionale non potato e creeremo un insieme di 500 alberi decisionali adattati su campioni di bootstrap differenti del dataset di addestramento:

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=None,
...                               random_state=1)
>>> bag = BaggingClassifier(base_estimator=tree,
...                         n_estimators=500,
...                         max_samples=1.0,
...                         max_features=1.0,
...                         bootstrap=True,
...                         bootstrap_features=False,
...                         n_jobs=1,
...                         random_state=1)
```

Poi calcolaremo la valutazione di accuratezza della previsione sul dataset di addestramento di test, per confrontare le prestazioni del classificatore di bagging con le prestazioni di un unico albero decisionale non potato:

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 1.000/0.833
```

Sulla base dei valori di accuratezza che abbiamo stampato eseguendo il frammento di codice precedente, l'albero decisionale non potato prevede correttamente tutte le etichette delle classi dei campioni di addestramento; tuttavia,

l'accuratezza sensibilmente inferiore nel campione di test indica un'elevata varianza (overfitting) del modello:

```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print('Bagging train/test accuracies %.3f/%.3f'
...       % (bag_train, bag_test))
Bagging train/test accuracies 1.000/0.896
```

Sebbene l'accuratezza di addestramento dell'albero decisionale e del classificatore di bagging siano simili sul set di addestramento (in entrambi i casi 1.0), possiamo vedere che il classificatore di bagging ha prestazioni di generalizzazione leggermente migliori, se stimate tramite il set di test. Quindi confrontiamo le regioni decisionali fra l'albero decisionale e il classificatore bagging:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, bag],
...                         ['Decision Tree', 'Bagging']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red', marker='o')
...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel(Alcohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...          s=Hue',
...          ha='center', va='center', fontsize=12)
>>> plt.show()
```

Come possiamo vedere nel grafico rappresentato nella Figura 7.8, il confine decisionale lineare dell'insieme sembra adattarsi meglio alla situazione rispetto a quello dell'albero decisionale con tre nodi di profondità.

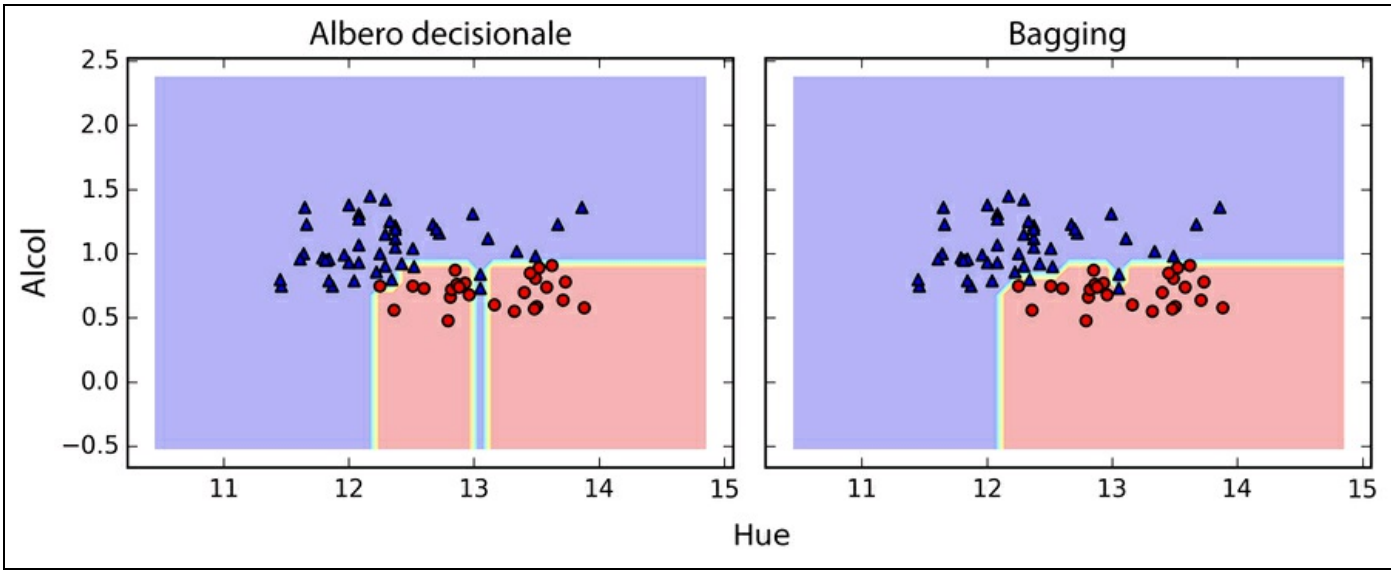


Figura 7.8

Abbiamo osservato solo un semplice esempio di bagging. Nella pratica, compiti di classificazione più complessi e dimensionalità più elevate dei dataset possono con facilità portare gli alberi decisionali a una condizione di overfitting e questi sono i casi in cui l'algoritmo di bagging può esprimere tutta la propria forza. Infine, dobbiamo notare che tale algoritmo può essere un approccio efficace per ridurre la varianza di un modello. Tuttavia, anche la tecnica di bagging è inefficace per ridurre il bias del modello, motivo per cui può essere meglio scegliere un insieme di classificatori con basso bias, per esempio degli alberi decisionali non potati.

Sfruttare i sistemi di apprendimento deboli tramite un boost adattativo

In questo paragrafo dedicato ai metodi d'insieme, parleremo di amplificazione (*boosting*) concentrandoci principalmente sull'implementazione più comune, *AdaBoost* (abbreviazione di *boost adattativo*).

NOTA

L'idea su cui si basa AdaBoost è stata formulata da Robert Schapire nel 1990 (R. E. Schapire, *The Strength of Weak Learnability*, in "Machine learning", 5(2):197–227, 1990). Dopo che Robert Schapire e Yoav Freund hanno presentato l'algoritmo AdaBoost nei Proceedings of the Thirteenth International Conference (ICML 1996), AdaBoost è diventato uno dei metodi d'insieme più ampiamente utilizzati negli anni a seguire (Y. Freund, R. E. Schapire et al. *Experiments with a New Boosting Algorithm*, in "ICML", volume 96, pp. 148–156, 1996). Nel 2003, Freund e Schapire hanno ricevuto il *Goedel Prize* per la loro opera rivoluzionaria, un premio prestigioso per le pubblicazioni più eminenti nel campo della computer science.

Nel boosting, l'insieme è costituito da classificatori base molto semplici, chiamati anche *sistemi di apprendimento deboli* (*weak learner*), i quali sono dotati di un debole vantaggio prestazionale rispetto alla scelta casuale. Un tipico esempio di sistema di apprendimento debole potrebbe essere un "moncone" di un albero decisionale. Il concetto su cui si basa il boosting è quello di concentrarsi sui campioni di addestramento difficili da classificare, ovvero far sì che i sistemi di apprendimento deboli agiscano sui campioni di apprendimento mal classificati, con lo scopo di migliorare le prestazioni dell'insieme. Al contrario del bagging, la formulazione iniziale del boosting, l'algoritmo utilizza sottoinsiemi casuali dei campioni di addestramento, tratti da un dataset di addestramento e senza reinserimento. La procedura di boosting originale può essere riepilogata in quattro passi.

1. Estrarre senza reinserimento dal set di addestramento D un sottoinsieme casuale dei campioni di addestramento d_1 per addestrare un sistema di apprendimento debole C_1.
2. Estrarre senza reinserimento dal set di addestramento un secondo sottoinsieme di addestramento casuale d_2 e aggiungere un 50% dei campioni che sono stati precedentemente classificati in modo errato, in modo da addestrare un sistema di apprendimento debole C_2.

3. Trovare i campioni di addestramento d_3 nel set di addestramento D in cui C_1 e C_2 non concordano, in modo da addestrare un terzo sistema di apprendimento debole C_3.
4. Combinare i risultati dei sistemi di apprendimento deboli C_1, C_2 e C_3tramite un voto a maggioranza.

Come discusso da Leo Breiman (L. Breiman, *Bias, Variance, and Arcing Classifiers*, 1996), il boosting può portare a una riduzione del bias e anche della varianza rispetto ai modelli a bagging. In pratica, però, gli algoritmi di boosting come AdaBoost sono noti anche per la loro elevata varianza, ovvero la tendenza a un overfitting sui dati di addestramento (G. Raetsch, T. Onoda e K. R. Mueller, *An Improvement of Adaboost to Avoid Overfitting*, in "Proc. of the Int. Conf. on Neural Information Processing", Citeseer, 1998).

Al contrario della procedura di boosting originale descritta qui, AdaBoost utilizza un set di addestramento completo per addestrare i sistemi di apprendimento deboli, dove i campioni di addestramento vengono ripesati a ogni iterazione, in modo da costruire un classificatore più efficace, che impara dagli errori dei precedenti sistemi di apprendimento deboli dell'insieme. Prima di approfondire i dettagli dell'algoritmo, esaminiamo la Figura 7.9, per avere una migliore idea dei concetti su cui si basa AdaBoost.

Esaminiamo l'illustrazione del funzionamento di AdaBoost passo dopo passo. Partiamo dal riquadro 1, che rappresenta un set di addestramento per una classificazione binaria, dove tutti i campioni di addestramento ricevono pesi uguali. Sulla base di questo set di addestramento, descriviamo un moncone decisionale (rappresentato come una linea tratteggiata) che tenta di classificare i campioni delle due classi (triangoli e cerchi) il meglio che sia possibile, minimizzando la funzione di costo (o la valutazione di impurità nel caso speciale di insiemi di alberi decisionali). Al turno successivo (riquadro 2), assegniamo un peso maggiore ai due campioni che in precedenza erano stati classificati erroneamente (cerchi). Inoltre riduciamo il peso dei campioni correttamente classificati. Il moncone decisionale successivo ora si concentrerà di più sui campioni di addestramento che hanno pesi maggiori, ovvero su quelli che, si suppone, sono difficili da classificare. Il sistema di apprendimento debole rappresentato nel riquadro 2 sbaglia a classificare tre diversi campioni della classe dei cerchi, ai quali viene pertanto assegnato un peso maggiore, come si può vedere nel riquadro 3. Supponendo che il nostro insieme AdaBoost consista solo di tre turni di boosting, possiamo combinare i tre sistemi di

apprendimento deboli addestrati tramite diversi sottoinsiemi di addestramento, ripesati con un voto a maggioranza pesato, come si può vedere nel riquadro 4.

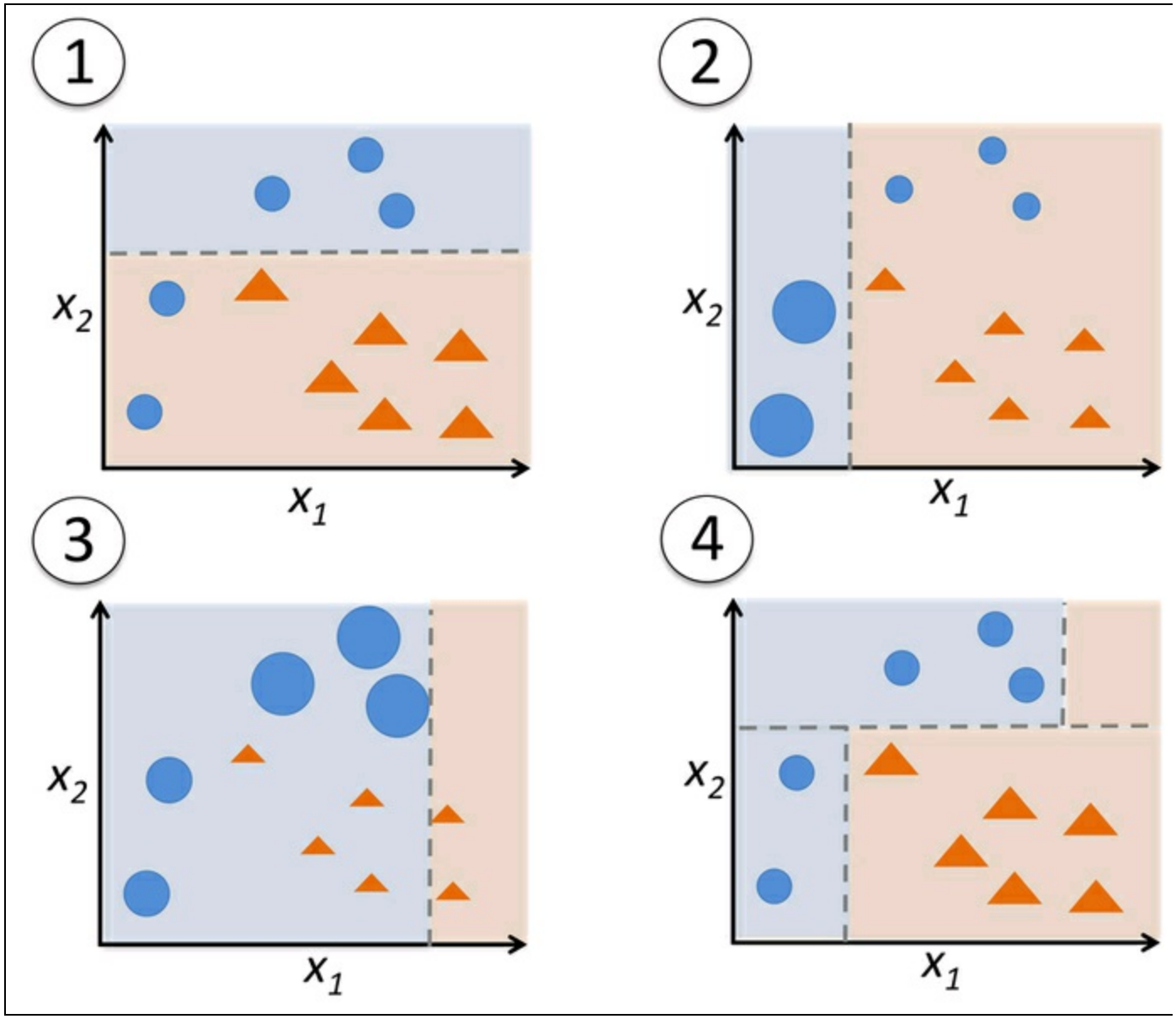


Figura 7.9

Ora che conosciamo meglio i concetti su cui si basa AdaBoost, esaminiamo in modo più dettagliato l'algoritmo, utilizzando lo pseudocodice. Per chiarezza, denoteremo la moltiplicazione fra elementi con il simbolo $(\times)$ e il prodotto fra due vettori con il simbolo $(\cdot)$. I passi sono i seguenti.

1. Impostare il vettore dei pesi $\boldsymbol{w}$ con pesi uniformi dove $\sum_i w_i = 1$.
2. Per $_j$ in $_m$ turni di boosting, ripetere i seguenti passi.
3. Addestrare un sistema di apprendimento debole pesato:
   $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w}$

4. Predire le etichette delle classi:
$$\hat{y} = \text{predict}\left(C_j, \boldsymbol{X}\right)$$
5. Calcolare il tasso d'errore pesato:
$$\varepsilon = \boldsymbol{w} \cdot \left(\hat{\boldsymbol{y}} == \boldsymbol{y}\right)$$
6. Calcolare il coefficiente:
$$\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$$
7. Aggiornare i pesi:
$$\boldsymbol{w} := \boldsymbol{w} \times \exp\left(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y}\right)$$
8. Normalizzare i pesi in modo che abbiano somma 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
9. Calcolare la previsione finale:
$$\hat{\boldsymbol{y}} = \left(\sum_{j=1}^{m} \left(\boldsymbol{\alpha}_j \times \text{predict}\left(C_j, \boldsymbol{X}\right)\right) > 0\right).$$

Notate che l'espressione $(\hat{\boldsymbol{y}} == \boldsymbol{y})$ nel Passo 5 fa riferimento a un vettore di 1 e 0, dove un 1 indica una previsione corretta e uno 0 indica una previsione errata.

Sebbene l'algoritmo AdaBoost sembra essere piuttosto semplice, esaminiamo un esempio più concreto utilizzando un set di addestramento costituito da 10 campioni di addestramento, come illustrato dalla Tabella 7.1.

Tabella 7.1

Indici campioni	x	y	Pesi	ŷ(x <=3.0)?	Corretto?	Pesi aggiornati
11	1.0	-1	0.1	-1	Sì	0.072
12	2.0	-1	0.1	-1	Sì	0.072
13	3.0	-1	0.1	-1	Sì	0.072
14	4.0	-1	0.1	-1	Sì	0.072
15	5.0	-1	0.1	-1	Sì	0.072
16	6.0	-1	0.1	-1	Sì	0.072
17	7.0	-1	0.1	-1	No	0.167
18	8.0	-1	0.1	-1	No	0.167
19	9.0	-1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Sì	0.072

La prima colonna della tabella rappresenta gli indici dei campioni di addestramento, da 1 a 10. Nella seconda colonna troviamo i valori della caratteristica dei singoli campioni, supponendo che si tratti di un dataset

monodimensionale. La terza colonna mostra l'etichetta della classe, y_i, per ciascun campione di addestramento x_i, dove $y_i \in \{1, -1\}$. I pesi iniziali sono riportati nella quarta colonna; inizializziamo i pesi in modo uniforme e normalizziamoli in modo che abbiano somma 1. Nel caso dei 10 campioni del set di addestramento, assegnamo pertanto 0.1 a ciascun peso w_i del vettore dei pesi $\boldsymbol{w}$. La quinta colonna mostra le etichette delle classi previste, $\hat{y}$, supponendo che il nostro criterio di discriminazione sia $x \leq 3.0$. L'ultima colonna della tabella mostra i pesi aggiornati sulla base delle regole di aggiornamento che abbiamo definito nello pseudocodice.

Poiché il calcolo degli aggiornamenti dei pesi può sembrare, inizialmente, piuttosto complesso, svolgeremo questo calcolo passo dopo passo. Iniziamo calcolando il tasso d'errore dei pesi ε come descritto nel Passo 5:

$$\varepsilon = 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 1 + 0.1\times 1 + 0.1\times 1 + 0.1\times 0 = \frac{3}{10} = 0.3$$

Poi calcoliamo il coefficiente α_j (rappresentato nel Passo 6), che verrà poi utilizzato nel Passo 7 per aggiornare i pesi e anche per determinare i pesi nella previsione con voto a maggioranza (Passo 10):

$$\alpha_j = 0.5 \log\left(\frac{1-\varepsilon}{\varepsilon}\right) \approx 0.424$$

Dopo aver calcolato il coefficiente α_j, possiamo aggiornare il vettore dei pesi utilizzando la seguente equazione:

$$\boldsymbol{w} := \boldsymbol{w} \times \exp\left(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y}\right)$$

Qui, $\hat{\boldsymbol{y}} \times \boldsymbol{y}$ è una moltiplicazione fra elementi, fra i vettori delle etichette delle classi previste e reali. Pertanto, se una previsione $\hat{y}_i$ è corretta, $\hat{y}_i \times y_i$ avrà segno positivo, in modo da decrementare il peso i-esimo, in quanto anche α_j è un numero positivo:

$$0.1 \times \exp\left(-0.424 \times 1 \times 1\right) \approx 0.066$$

Analogamente, ci troveremo a ridurre il peso i-esimo se $\hat{y}_i$ ha previsto l'etichetta in modo errato:

$$0.1 \times \exp\left(-0.424 \times 1 \times (-1)\right) \approx 0.153$$

o anche così:

$$0.1 \times \exp\left(-0.424 \times (-1) \times (1)\right) \approx 0.153$$

Dopo aver aggiornato ciascun peso nel vettore dei pesi, normalizziamo i pesi, in modo che la somma dia sempre 1 (Passo 8):

$$\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i w_i}$$

Qui:

$$\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$$

Pertanto, ogni peso che corrisponde a un campione correttamente classificato verrà ridotto rispetto al suo valore iniziale 0.1 a

$$0.066 / 0.914 \approx 0.072$$

per il turno successivo di boosting. Analogamente, i pesi per i campioni classificati in modo errato verranno incrementati da 0.1 a

$$0.153 / 0.914 \approx 0.167$$

Questa è la sostanza dell'algoritmo AdaBoost. Saltando a una parte più pratica, descriviamo un classificatore d'insieme AdaBoost realizzato tramite scikit-learn. Utilizzeremo lo stesso sottoinsieme Wine che abbiamo impiegato nel paragrafo precedente per addestrare il meta-classificatore di bagging. Tramite l'attributo `base_estimator`, addestreremo `AdaBoostClassifier` su 500 tronconi di alberi decisionali:

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=None,
...                               random_state=0)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          learning_rate=0.1,
...                          random_state=0)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 0.845/0.854
```

Come possiamo vedere, il troncone di albero decisionalo sembra soffrire di overfitting sui dati di addestramento, al contrario dell'albero decisionale non potato che abbiamo visto nel paragrafo precedente:

```
>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('AdaBoost train/test accuracies %.3f/%.3f'
...       % (ada_train, ada_test))
AdaBoost train/test accuracies 1.000/0.875
```

Come possiamo vedere, il modello AdaBoost prevede correttamente tutte le etichette delle classi del set di addestramento e mostra anche prestazioni leggermente migliori sul set di test rispetto al moncone di albero. Tuttavia, vediamo anche che abbiamo introdotto una maggiore varianza, con il nostro tentativo di ridurre il bias del modello.

Anche se abbiamo utilizzato un altro semplice esempio per scopi dimostrativi, possiamo vedere che le prestazioni del classificatore AdaBoost sono leggermente migliorate rispetto al moncone decisionale, e hanno fornito punteggi di accuratezza molto simili a quelli del classificatore bagging che abbiamo addestrato nel paragrafo precedente. Tuttavia, dobbiamo notare che è considerata una cattiva abitudine selezionare un modello sulla base di un utilizzo ripetuto del set di test. La stima delle prestazioni di generalizzazione può sembrare troppo ottimistica, come abbiamo descritto in dettaglio nel Capitolo 6,*Valutazione dei modelli e ottimizzazione degli iperparametri*.

Infine, controlliamo l'aspetto delle regioni decisionali:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, ada],
...                         ['Decision Tree', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red',
...                        marker='o')
... axarr[idx].set_title(tt)
... axarr[0].set_ylabel('Alcohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...          s=Hue',
```

```
...         ha='center',
...         va='center',
...         fontsize=12)
>>> plt.show()
```

Osservando le regioni decisionali, possiamo vedere che il confine decisionale del modello AdaBoost è molto più complesso rispetto a quello del moncone decisionale. Inoltre, notiamo che il modello AdaBoost separa lo spazio delle caratteristiche in modo molto simile al classificatore bagging che abbiamo addestrato nel paragrafo precedente (Figura 7.10).

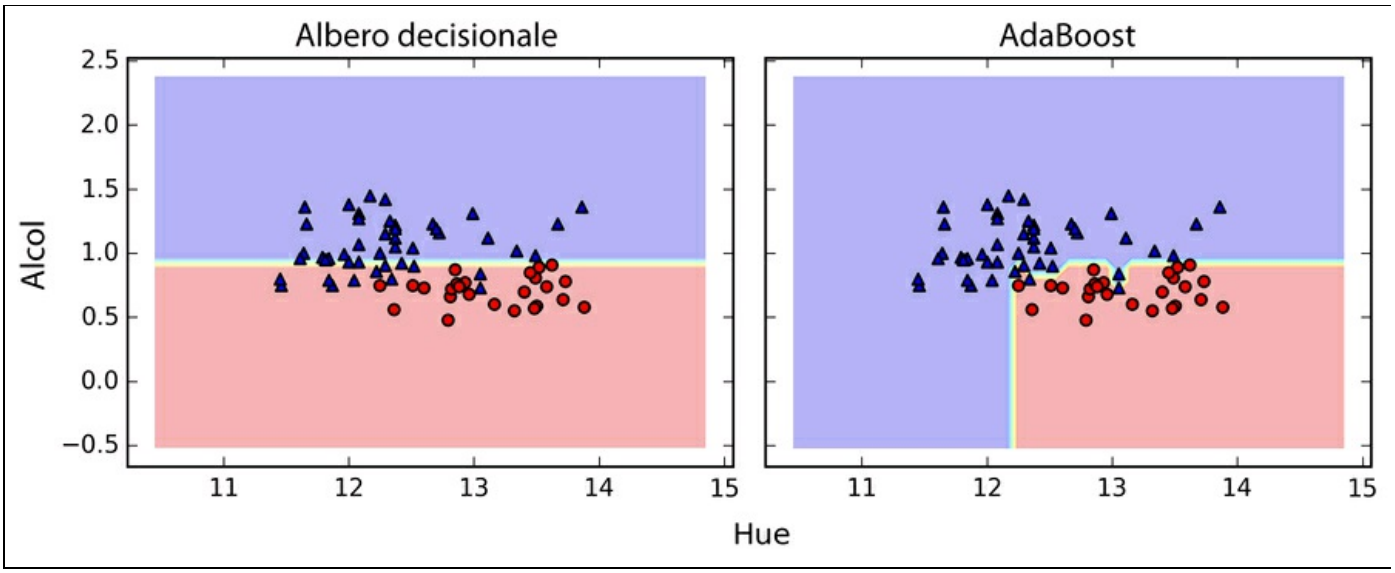


Figura 7.10

Come note conclusive sulle tecniche d'insieme, vale la pena di notare che l'apprendimento d'insieme incrementa la complessità computazionale rispetto ai singoli classificatori. Nella pratica, dobbiamo riflettere attentamente se intendiamo pagare il prezzo di questo maggiore costo computazionale per ottenere solo un relativamente modesto miglioramento delle prestazioni predittive.

Un esempio spesso citato di questo compromesso è quello del famoso *$1 Million Netflix Prize*, che è stato vinto utilizzando tecniche d'insieme. I dettagli relativi all'algoritmo sono stati pubblicati in A. Toescher, M. Jahrer e R. M. Bell, *The Bigchaos Solution to the Netfl ix Grand Prize*, in "Netflix prize documentation", 2009 (disponibile all'indirizzo http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf). Dopo che il team vincitore ha ricevuto il premio in denaro di un milione di dollari, Netflix non ha mai implementato il loro modello a causa della sua complessità, che lo rendeva inutilizzabile nelle applicazioni pratiche. Ecco quali sono state le loro parole (http://techblog.netflix.com/2012/04/netflixrecommendations-beyond-5-stars.html):

> […] l'incremento aggiuntivo dell'accuratezza che abbiamo misurato non sembra giustificare l'impegno tecnico necessario per impiegarlo in un ambiente di produzione.

Riepilogo

In questo capitolo abbiamo esaminato alcune delle tecniche più note e ampiamente utilizzate di apprendimento d'insieme. I metodi d'insieme combinano modelli di classificazione differenti per annullare i loro singoli punti deboli, il che spesso produce modelli più stabili e di prestazioni superiori, che sono molto interessanti per le applicazioni industriali, così come per le competizioni nell'ambito del machine learning.

All'inizio del capitolo abbiamo implementato un `MajorityVoteClassifier` in Python, che ci ha consentito di combinare algoritmi di classificazione differenti. Abbiamo poi parlato del bagging, una tecnica utile per ridurre la varianza di un modello, estraendo campioni di bootstrap casuali dal set di addestramento e combinando dei classificatori addestrati singolarmente tramite un voto a maggioranza. Poi abbiamo parlato di AdaBoost, un algoritmo che si basa su sistemi di apprendimento debole che imparano dai propri errori.

Nei capitoli precedenti abbiamo trattato vari algoritmi di apprendimento, abbiamo parlato di ottimizzazione e di tecniche di valutazione. Nel prossimo capitolo esamineremo una particolare applicazione degli algoritmi di machine learning: l'analisi del sentiment, che è diventato certamente un argomento interessante nell'epoca di Internet e dei social media.

Tecniche di machine learning per l'analisi del sentiment

Nell'era di Internet e dei social media, le opinioni, le recensioni e i consigli delle persone sono diventati una risorsa preziosa per la scienza politica e il commercio. Grazie alle moderne tecnologie, siamo ora in grado di raccogliere e analizzare tali dati in modo più efficiente. In questo capitolo approfondiremo un sottocampo dell'elaborazione del linguaggio naturale (*NLP – natural language processing*) chiamato *analisi del sentiment* e impareremo a utilizzare gli algoritmi di machine learning per classificare i documenti sulla base dell'atteggiamento dell'autore. Ecco alcuni degli argomenti che tratteremo in questi paragrafi.

- Ripulitura e preparazione dei dati testuali.
- Creazione di vettori delle caratteristiche a partire dai documenti testuali.
- Addestramento di un modello di machine learning per classificare le recensioni positive e negative di un film.
- Utilizzo di grossi dataset testuali tramite l'apprendimento *out-of-core*.

Accedere al dataset delle recensioni dei film di IMDb

L'analisi del sentiment, chiamata anche *opinion mining*, è una popolare branca del campo ben più ampio dell'elaborazione del linguaggio naturale (NLP). Il suo scopo è quello di analizzare la *polarità* dei documenti. Un compito molto diffuso nell'analisi del sentiment è la classificazione dei documenti sulla base delle opinioni e delle emozioni espresse dagli autori a proposito di un determinato argomento.

In questo capitolo lavoreremo con un grosso dataset di recensioni di film tratto da *IMDb* (*Internet Movie Database*) che è stato raccolto da Maas et al. (A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng e C. Potts, *Learning Word Vectors for Sentiment Analysis*, in "Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies", pp. 142–150, Portland, Oregon, USA, Giugno 2011, Association for Computational Linguistics). Il dataset è costituito da 50.000 recensioni di film, che sono state etichettate come *positive* o *negative*. Si intende che un film è stato valutato positivamente se ha ricevuto più di sei stelle su IMDb e negativamente se ha ricevuto meno di cinque stelle. Nei prossimi paragrafi, impareremo a estrarre informazioni utili da un sottoinsieme di queste recensioni, per costruire un modello che sia in grado di prevedere se un determinato recensore ha apprezzato o meno un film.

Un archivio compresso del dataset di recensioni sui film (84.1 MB) può essere scaricato da http://ai.stanford.edu/~amaas/data/sentiment/.

- Se utilizzate Linux o Mac OS X, potete aprire una nuova finestra del terminale, poi utilizzare il comando `cd` per accedere alla directory `download` ed eseguire il comando `tar -zxfaclImdb_v1.tar.gz` per decomprimere il dataset.
- Se invece utilizzate Windows, potete scaricare un sistema di archiviazione gratuito come 7-Zip (http://www.7-zip.org) per estrarre i file dall'archivio scaricato.

Dopo aver estratto con successo il dataset, potete assemblare i singoli documenti testuali dall'archivio scaricato decompresso per formare un unico file CSV. Nella prossima sezione di codice, leggeremo le recensioni dei film all'interno di un oggetto pandas `DataFrame`, operazione che può richiedere anche una decina di minuti su un computer desktop standard. Per rappresentare i progressi e il tempo stimato

prima del completamento, utilizzeremo il pacchetto *PyPrind* (Python Progress Indicator, https://pypi.python.org/pypi/PyPrind/) che l'autore del libro ha sviluppato alcuni anni fa proprio per questi scopi. PyPrind può essere installato eseguendo il comando `pip install pyprind`.

```
>>> import pyprind
>>> import pandas as pd
>>> import os
>>> pbar = pyprind.ProgBar(50000)
>>> labels = {'pos':1, 'neg':0}
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...    for l in ('pos', 'neg'):
...        path ='./aclImdb/%s/%s' % (s, l)
...        for file in os.listdir(path):
...            with open(os.path.join(path, file), 'r') as infile:
...                txt = infile.read()
...            df = df.append([[txt, labels[l]]], ignore_index=True)
...            pbar.update()
>>> df.columns = ['review', 'sentiment']
0%                          100%
[##############################] | ETA[sec]: 0.000
Total time elapsed: 725.001 sec
```

Eseguendo il codice precedente, dobbiamo innanzitutto inizializzare una nuova barra di progressione (l'oggetto `pbar`) con 50.000 iterazioni, il numero di documenti che dobbiamo leggere. Utilizzando cicli `for` nidificati, abbiamo eseguito un'iterazione sulle sue directory `train` e `test` della directory principale `aclImdb` e abbiamo letto i singoli file di testo dalle subdirectory `pos` e `neg` che alla fine abbiamo aggiunto al `DataFrame df`, insieme a un'etichetta della classe intera (`1` per positivo e `0` per negativo).

Poiché le etichette delle classi in un dataset assemblato sono ordinate, dobbiamo mescolare `DataFrame` utilizzando la funzione `permutation` del modulo `np.random`. Questo sarà utile poi per suddividere il dataset nei set di addestramento e di test nei prossimi paragrafi, quando trarremo i dati direttamente dall'unità dischi locale. Per comodità, memorizzeremo anche il dataset di recensioni assemblato e mescolato sotto forma di file CSV:

```
>>> import numpy as np
>>> np.random.seed(0)
>>> df = df.reindex(np.random.permutation(df.index))
>>> df.to_csv('./movie_data.csv', index=False)
```

Poiché ci troveremo a utilizzare altre volte questo dataset più avanti in questo capitolo, confermiamo rapidamente di aver salvato con successo i dati nel formato corretto leggendo il file e stampando un estratto dei primi tre campioni:

```
>>> df = pd.read_csv('./movie_data.csv')
>>> df.head(3)
```

Se state eseguendo gli esempi di codice in IPython Notebook, a questo punto dovreste vedere i primi tre campioni del dataset, come nella Tabella 8.1.

Tabella 8.1

	Recensione	Sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

Introduzione al modello bag-of-words

Ricordiamo, dal Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione*, che dobbiamo convertire i dati categorici, per esempio il testo o le parole, in una forma numerica prima di poterli passare a un algoritmo di apprendimento automatico. In questo paragrafo introdurremo il modello *bag-of-words*, che ci consente di rappresentare il testo sotto forma di vettori numerici delle caratteristiche. L'idea su cui si basa il modello bag-of-words è piuttosto semplice e può essere riassunta nel seguente modo.

1. Creiamo un *vocabolario* di *token* univoci (per esempio, parole) utilizzando un intero insieme di documenti.
2. Costruiamo da ciascun documento un vettore delle caratteristiche, che contiene il conteggio della frequenza delle parole all'interno di quello specifico documento.

Poiché le parole univoche in ciascun documento rappresentano solo un piccolo sottoinsieme di tutte le parole del vocabolario bag-of-words, i vettori delle caratteristiche saranno costituiti principalmente da valori 0, ovvero si tratta di vettori *sparsi*. Non preoccupatevi se al momento tutto ciò vi sembra un po' troppo astratto: nei prossimi paragrafi esamineremo passo dopo passo il processo di creazione di un semplice modello bag-of-words.

Trasformazione delle parole in vettori di caratteristiche

Per costruire un modello bag-of-words basato sul conteggio delle parole nei rispettivi documenti, possiamo utilizzare la classe `CountVectorizer` implementata in scikit-learn. Come vedremo nella prossima sezione di codice, la classe `CountVectorizer` prende un array di dati testuali, che possono essere documenti o semplici frasi, e costruisce per noi il modello bag-of-words:

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array([
...       'The sun is shining',
...       'The weather is sweet',
...       'The sun is shining and the weather is sweet'])
>>> bag = count.fit_transform(docs)
```

Richiamando il metodo fit_transform su CountVectorizer, abbiamo appena costruito il vocabolario del modello bag-of-words e abbiamo trasformato le tre frasi seguenti in vettori sparsi di caratteristiche:

1. The sun is shining
2. The weather is sweet
3. The sun is shining and the weather is sweet

Ora stampiamo il contenuto del vocabolario per capire meglio i concetti sottostanti:

```
>>> print(count.vocabulary_)
{'the': 5, 'shining': 2, 'weather': 6, 'sun': 3, 'is': 1, 'sweet': 4, 'and': 0}
```

Come possiamo vedere dall'esecuzione del comando precedente, il vocabolario è conservato in un dizionario Python, che mappa parole univoche su indici interi. Poi stampiamo i vettori delle caratteristiche che abbiamo appena creato:

```
>>> print(bag.toarray())
[[0 1 1 1 0 1 0]
 [0 1 0 0 1 1 1]
 [1 2 1 1 1 2 1]]
```

Ogni posizione indice dei vettori delle caratteristiche rappresentati corrisponde ai valori interi che sono conservati come elementi del dizionario nel vocabolario CountVectorizer. Per esempio, la prima caratteristica alla sezione con indice 0 riguarda il conteggio della parola and, che si verifica solo nell'ultimo documento e la parola is alla posizione con indice 1 (la seconda caratteristica nei vettori del documento) si presenta in tutte e tre le frasi. Questi valori nei vettori delle caratteristiche sono chiamati anche *frequenze grezze dei termini* o *raw term frequencies*: *tf (t,d)*, il numero di volte in cui il termine *t* si presenta nel documento *d*.

Approfondimento

La sequenza degli elementi nel modello bag-of-words che abbiamo appena creato è chiamata anche modello *1-gram* o *unigram*: ogni elemento o token del vocabolario rappresenta un'unica parola. Più in generale, nell'elaborazione del linguaggio naturale, le sequenze contigue di elementi (parole, lettere o simboli) sono chiamate *n-gram*. La scelta del numero *n* in un modello n-gram dipende dalla specifica applicazione; per esempio, uno studio di Kanaris ha rilevato che gli n-gram di dimensioni 3 e 4 offrono buone prestazioni nel filtraggio anti-spam dei messaggi di posta elettronica (Ioannis Kanaris, Konstantinos Kanaris, Ioannis Houvardas e Efstathios Stamatatos, *Words vs Character N-Grams for Anti-Spam Filtering*, in "International Journal on Artificial Intelligence Tools", 16(06):1047–1067, 2007). Per riepilogare il concetto della rappresentazione a n-gram, le rappresentazioni 1-gram e 2-gram del nostro primo documento "the sun is shining" verrebbero costruite nel seguente modo:

- *1-gram*: "the", "sun", "is", "shining"
- *2-gram*: "the sun", "sun is", "is shining"

La classe CountVectorizer in scikit-learn ci consente di utilizzare modelli n-gram differenti tramite il parametro ngram_range. Mentre per default viene utilizzata una rappresentazione 1-gram, potremmo passare a una rappresentazione 2-gram inizializzando una nuova istanza di CountVectorizer con ngram_range=(2,2).

Valutazione della rilevanza delle parole in base alla frequenza (inversa) dei termini nel documento

Quando analizziamo dei dati testuali, spesso incontriamo parole che si presentano in più documenti di entrambe le classi. Queste parole che si presentano frequentemente in genere non contengono informazioni utili o discriminative. In questo paragrafo impareremo a utilizzare una tecnica utile chiamata frequenza inversa dei termini e chiamata *tf-idf* (*term frequency-inverse document frequency*), la quale può essere utilizzata per ridurre l'incidenza di quelle parole che si presentano frequentemente nei vettori delle caratteristiche. La tecnica tf-idf può essere definita come il prodotto della frequenza del termine e della frequenza inversa del documento:

$$\text{tf-idf}\left(\text{t,d}\right) = tf\left(t,d\right) \times \text{idf}\left(\text{t,d}\right)$$

Qui, *tf(t, d)* è la frequenza del termine, che è stata introdotta nel paragrafo precedente; la frequenza inversa nel documento *idf(t, d)* può essere calcolata come:

$$\text{idf}\left(\text{t,d}\right) = log\frac{n_d}{1+\text{df}\left(\text{d,t}\right)}$$

dove n_d è il numero totale dei documenti e *df(d, t)* è il numero di documenti *d* che contengono il termine *t*. Notate che l'aggiunta della costante 1 al denominatore è opzionale e ha lo scopo di assegnare un valore diverso da 0 ai termini che si presentano in tutti campioni di addestramento. Il logaritmo viene utilizzato per evitare che ai termini poco frequenti nel documento venga assegnato un peso eccessivo.

Scikit-learn implementa un altro trasformer, TfidfTransformer, che prende come input le frequenze grezze dei termini da CountVectorizer e le trasforma in tf-idfs:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tfidf = TfidfTransformer()
>>> np.set_printoptions(precision=2)
>>> print(tfidf.fit_transform(count.fit_transform(docs)).toarray())
[[ 0.   0.43  0.56  0.56  0.   0.43  0.  ]
 [ 0.   0.43  0.   0.   0.56  0.43  0.56]
 [ 0.4  0.48  0.31  0.31  0.31  0.48  0.31]]
```

Come abbiamo visto nei paragrafi precedenti, la parola is ha la frequenza più elevata nel terzo documento, dove è la parola che si presenta più frequentemente. Tuttavia, dopo aver trasformato lo stesso vettore di caratteristiche in tf-idfs, vediamo che la parola is ora ottiene un tf-idf piuttosto limitato (0.31) come nel terzo documento, poiché è contenuta anche nel primo e nel secondo documento e pertanto è improbabile che contenga informazioni utili e discriminati.

Tuttavia, se calcolasimo manualmente i valori tf-idfs dei singoli termini dei vettori delle caratteristiche, noteremmo che TfidfTransformer calcola i pesi tf-idfs in modo leggermente differente rispetto alle equazioni testuali *standard* che abbiamo definito in precedenza. L'equazione per idf che è stata implementata in scikit-learn è:

$$\text{idf}\left(\text{t,d}\right) = log\frac{1+n_d}{1+\text{df}\left(\text{d,t}\right)}$$

L'equazione tf-idf che è stata implementata in scikit-learn è:

$$\text{tf-idf}\left(\text{t,d}\right) = tf\left(\text{t,d}\right)\times\left(\text{idf}\left(\text{t,d}\right)+1\right)$$

Anche se in genere si normalizzano le frequenze grezze dei termini prima di calcolare i valori tf-idfs, TfidfTransformer normalizza direttamente tf-idfs. Per impostazione predefinita (norm='l2'), TfidfTransformer di scikit-learn applica la normalizzazione L2, che restituisce un vettore di lunghezza unitaria, dividendo un vettore delle caratteristiche non normalizzato *v* per il suo L2:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}} = \frac{v}{\left(\sum_{i=1}^{n} v_i^2\right)^{1/2}}$$

Per fissare in mente il funzionamento di TfidfTransformer, esaminiamo un esempio e calcoliamo il valore *tf-idf* della parola is nel terzo documento.

La parola is ha una frequenza pari a 2 (tf = 2) nel terzo documento e la frequenza nei documenti di questo termine è 3, in quanto il termine is si presenta in tutti e tre i documenti (df = 3). Pertanto possiamo calcolare il valore *idf* nel seguente modo:

$$idf\left("\text{is}", \text{d3}\right) = \log\frac{1+3}{1+3} = 0$$

Ora, per calcolare il valore *tf-idf*, dobbiamo semplicemente aggiungere 1 alla frequenza inversa nel documento e moltiplicare il tutto per la frequenza del termine:

$$\text{tf-idf}\left("is", \text{d3}\right) = 2 \times \left(0+1\right) = 2$$

Se ripettessimo questi calcoli per tutti i termini del terzo documento, otterremo il seguente vettore *tf-idf*: [1.69, 2.00, 1.29, 1.29, 1.29, 2.00 e 1.29]. Tuttavia, notiamo che i valori di questo vettore delle caratteristiche sono differenti rispetto ai valori che abbiamo ottenuto dal trasformer `TfidfTransformer` che abbiamo utilizzato in precedenza. Il passo finale che ci manca nel calcolo di questo *tf-idf* è la normalizzazione L2, che possiamo applicare nel seguente modo:

$$\text{tf-idf}\left("is", \text{d3}\right)_{norm} \frac{\left[1.69,\ 2.00,\ 1.29,\ 1.29,\ 1.29,\ 2.00,\ 1.29\right]}{\sqrt{1.69^2 + 2.00^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.00^2 + 1.29^2}}$$
$$= \left[0.40,\ 0.48,\ 0.31,\ 0.31,\ 0.31,\ 0.48,\ 0.31\right]$$

Come possiamo vedere, i risultati ora corrispondono ai risultati restituiti dal trasformer `TfidfTransformer` di scikit-learn. Poiché ora sappiamo come vengono calcolati da scikit-learn i valori tf-idfs, cominciamo ad applicare questi concetti al dataset delle recensioni sui film.

Pulitura dei dati testuali

Nei paragrafi precedenti abbiamo parlato del modello bag-of-words, delle frequenze dei termini e di tf-idfs. Tuttavia, il passo più importante, prima ancora di costruire il modello bag-of-words, consiste nel ripulire i dati testuali, eliminando tutti i caratteri indesiderati. Per illustrare perché questo passo è importante, esaminiamo gli ultimi 50 caratteri del primo documento del dataset delle recensioni dei film dopo il mescolamento:

```
>>> df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

Come possiamo vedere, il testo contiene codice HTML, segni di punteggiatura e altri caratteri non letterali. Mentre il codice HTML non contiene molte informazioni semantiche utili, i segni di punteggiatura possono rappresentare informazioni

aggiuntive utili per determinati contesti di elaborazione del linguaggio naturale. Tuttavia, per semplicità, ora elimineremo tutti i segni di punteggiatura, mantenendovi solo i caratteri *emoticon*, come ":)", che in genere sono utili per l'analisi del sentiment. Per svolgere questo compito, utilizzeremo la libreria *regex* (*regular expression*) di Python, dedicata all'elaborazione delle espressioni regolari, ovvero:

```
>>> import re
>>> def preprocessor(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)
...     text = re.sub('[\W]+', ' ', text.lower()) + \
                ' '.join(emoticons).replace('-', '')
...     return text
```

Tramite la prima regex <[^>]*> della sezione di codice precedente, abbiamo tentato di eliminare l'intero codice HTML contenuto nelle recensioni dei film. Sebbene molti programmatori siano generalmente contrari all'uso delle espressioni regolari per analizzare il codice HTML, questa espressione dovrebbe essere sufficiente per *ripulire* questo specifico dataset. Dopo aver rimosso il codice HTML, abbiamo utilizzato un'espressione regolare leggermente più complessa per trovare le emoticon, che abbiamo conservato temporaneamente come `emoticons`. Poi abbiamo rimosso tutti i caratteri non alfabetici tramite l'espressione regolare `[\W]+`, quindi abbiamo convertito il testo in caratteri minuscoli e alla fine abbiamo aggiunto le emoticon, conservate temporaneamente, alla fine della stringa del documento, dopo l'elaborazione. Inoltre, per coerenza, abbiamo eliminato dalle emoticon il trattino che ne rappresenta il "naso", ovvero "-".

NOTA

Sebbene l'espressioni regolari offrano un approccio efficiente e comodo per la ricerca dei caratteri contenuti in una stringa, presentano anche una curva di apprendimento ripida. Sfortunatamente, una discussione approfondita delle espressioni regolari non rientra negli scopi di questo libro. Tuttavia potete trovare ottimi tutorial nel portale Google Developers all'indirizzo https://developers.google.com/edu/python/regular-expressions oppure potete consultare la documentazione ufficiale del modulo `re` di Python all'indirizzo https://docs.python.org/3.4/library/re.html.

Sebbene l'aggiunta dei caratteri delle emoticon alla fine del documento ripulito possa non sembrare l'approccio più elegante, l'ordine delle parole non conta nel nostro modello bag-of-words se il vocabolario è costituito solo da token di una parola. Ma prima di parlare della suddivisione dei documenti in singoli termini, parole o token, vediamo se il nostro preprocessore ha funzionato correttamente:

```
>>> preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
>>> preprocessor("</a>This :) is :( a test :-)!")
'this is a test :) :( :)'
```

Infine, poiché nel corso dei prossimi paragrafi faremo più volte uso dei dati costituiti da testo *ripulito*, applichiamo la funzione `preprocessor` a tutte le recensioni dei film del `DataFrame`:

```
>>> df['review'] = df['review'].apply(preprocessor)
```

Elaborazione dei documenti in token

Dopo aver preparato con successo il dataset delle recensioni sui film, dobbiamo pensare a suddividere i contenuti testuali nei singoli elementi. Un modo per *token*-izzare i documenti consiste nel suddividerli nelle singole parole, spezzando il documento ripulito in corrispondenza degli spazi:

```
>>> def tokenizer(text):
...     return text.split()
>>> tokenizer('runners like running and thus they run')
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

Nel contesto della tokenizzazione, un'altra tecnica utile si chiama *word stemming*, il processo di trasformazione di una parola nella sua forma radicale, che ci consentirà di aggregare le parole aventi la stessa radice. L'algoritmo originario di stemming è stato sviluppato da Martin F. Porter nel 1979 e per questo motivo è chiamato *algoritmo stemmer di Porter* (Martin F. Porter, *An algorithm for suffix stripping*, in "Program: electronic library and information systems", 14(3):130–137, 1980). Il Natural Language Toolkit di Python (NLTK, http://www.nltk.org) implementa l'algoritmo di stemming di Porter, che utilizzeremo nella prossima sezione di codice. Per installare NLTK, basta eseguire `pip install nltk`.

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
>>> def tokenizer_porter(text):
...     return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

NOTA

Sebbene NLTK non sia l'argomento del capitolo, è assolutamente consigliabile visitare il sito web di NLTK e anche la documentazione ufficiale di NLTK, liberamente disponibile all'indirizzo http://www.nltk.org/book/ qualora siate interessati ad applicazioni più avanzate di elaborazione del linguaggio naturale.

Utilizzando `PorterStemmer` del vecchio `nltk`, abbiamo modificato la nostra funzione `tokenizer` per ridurre le parole alla loro forma radicale, che è stata illustrata dal precedente semplice esempio quando la parola `running` è stata ridotta la sua radice `run`.

NOTA

Lo stemmer di Porter è probabilmente il più vecchio è il più semplice degli algoritmi di stemming. Fra gli altri algoritmi comunemente utilizzati vi sono il più recente *Snowball stemmer*

(Porter2 o “English” stemmer) o il *Lancaster stemmer* (Paice-Husk stemmer) che è più veloce ma anche aggressivo rispetto al stemmer di Porter. Questi algoritmi di stemming alternativi sono disponibili anche nel packaging NLTK (http://www.nltk.org/api/nltk.stem.html). Anche se l’operazione di stemming può produrre parole non reali, come thu (da thus) come illustrato dall’esempio precedente, una tecnica chiamata *lemmizzazione* mira a ottenere le forme canoniche (corrette dal punto di vista grammaticale) delle singole parole, i cosiddetti *lemmi*. Tuttavia, la lemmizzazione è più difficoltosa e costosa dal punto di vista computazionale rispetto allo stemming e, in pratica, si è osservato che l’uso dell’una o dell’altra tecnica ha un impatto limitato sulle prestazioni della classificazione del testo (Michal Toman, Roman Tesar e Karel Jezek, *Influence of word normalization on text classification*, in “Proceedings of InSciT”, pp. 354–358, 2006).

Prima di passare al prossimo paragrafo, in cui descriveremo un modello di machine learning utilizzando un modello bag-of-words, parliamo brevemente di un altro argomento utile chiamato *stop-word removal*. Le stop-word sono semplicemente quelle parole che sono estremamente comuni in ogni genere di testo e che, in genere, non comunicano particolari informazioni che possano essere utilizzate per distinguere classi di documenti. Fra gli esempi di stop-word [della lingua inglese, N.d.T.] vi sono *is, and, has* e così via. La rimozione delle stop-word può essere utile se si utilizzano frequenze di termini grezzi o normalizzate invece di tf-idfs, che già di per sé riduce il peso delle parole che si presentano più frequentemente.

Per eliminare le stop-word dalle recensioni dei film, utilizzeremo il set delle 127 stop-word della lingua inglese, disponibile nella libreria NLTK, che si può ottenere richiamando la funzione nltk.download:

```
>>> import nltk
>>> nltk.download('stopwords')
```

Dopo aver scaricato il set di stop-word, possiamo caricarlo e applicarlo nel seguente modo:

```
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>>  [w for w in tokenizer_porter('a runner likes running and runs a lot')[-10:] if w not in stop]
['runner', 'like', 'run', 'run', 'lot']
```

Addestramento di un modello a regressione logistica per la classificazione dei documenti

Ora addestreremo un modello a regressione logistica per classificare le recensioni dei film e suddividerle fra positive e negative. Innanzitutto divideremo il DataFrame dei documenti testuali ripuliti in 25.000 documenti di addestramento e 25.000 documenti di test:

```
>>> X_train = df.loc[:25000, 'review'].values
>>> y_train = df.loc[:25000, 'sentiment'].values
>>> X_test = df.loc[25000:, 'review'].values
>>> y_test = df.loc[25000:, 'sentiment'].values
```

Poi utilizzeremo l'oggetto GridSearchCV per trovare il set di parametri ottimale per il modello a regressione logistica, utilizzando la convalida incrociata stratificata a cinque parti:

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> tfidf = TfidfVectorizer(strip_accents=None,
...                         lowercase=False,
...                         preprocessor=None)
>>> param_grid = [{'vect__ngram_range': [(1,1)],
...               'vect__stop_words': [stop, None],
...               'vect__tokenizer': [tokenizer,
...                                   tokenizer_porter],
...               'clf__penalty': ['l1', 'l2'],
...               'clf__C': [1.0, 10.0, 100.0]},
...               {'vect__ngram_range': [(1,1)],
...               'vect__stop_words': [stop, None],
...               'vect__tokenizer': [tokenizer,
...                                   tokenizer_porter],
...               'vect__use_idf':[False],
...               'vect__norm':[None],
...               'clf__penalty': ['l1', 'l2'],
...               'clf__C': [1.0, 10.0, 100.0]}
...              ]
>>> lr_tfidf = Pipeline([('vect', tfidf),
...                      ('clf',
...                       LogisticRegression(random_state=0))])
>>> gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
...                            scoring='accuracy',
...                            cv=5, verbose=1,
...                            n_jobs=-1)
>>> gs_lr_tfidf.fit(X_train, y_train)
```

Quando abbiamo inizializzato l'oggetto GridSearchCV e la sua griglia di parametri utilizzando il codice precedente, ci siamo limitati solo ad alcune combinazioni di parametri, poiché il numero di vettori delle caratteristiche, così come l'ampiezza del vocabolario, possono rendere la ricerca a griglia piuttosto costosa dal punto di vista computazionale; utilizzando un normale computer desktop, la ricerca a griglia può richiedere una quarantina di minuti.

Nell'esempio di codice precedente, abbiamo sostituito il `CountVectorizer` e il `TfidfTransformer` del paragrafo precedente con `TfidfVectorizer`, che combina tali oggetti del trasformer. La nostra `param_grid` è costituita da due dizionari di parametri. Nel primo dizionario abbiamo utilizzato `TfidfVectorizer` con le sue impostazioni standard (`use_idf=True`, `smooth_idf=True` e `norm='l2'`) per calcolare i valori tf-idfs; nel secondo dizionario abbiamo impostato questi parametri a `use_idf=False`, `smooth_idf=False` e `norm=None`, in modo da addestrare un modello sulla base delle frequenze grezze dei termini. Inoltre, per il classificatore a regressione logistica, abbiamo addestrato i modelli utilizzando la regolarizzazione L2 e L1 tramite il parametro `penalty` e abbiamo confrontato varie intensità di regolarizzazione definendo un intervallo di valori per il parametro di regolarizzazione inversa, `C`.

Una volta terminata la ricerca a griglia, possiamo stampare il set dei parametri ottimali:

```
>>> print('Best parameter set: %s ' % gs_lr_tfidf.best_params_)
Best parameter set: {'clf__C': 10.0, 'vect__stop_words': None, 'clf__penalty': 'l2', 'vect__tokenizer': <function tokenizer at 0x7f6c704948c8>,
'vect__ngram_range': (1, 1)}
```

Come possiamo vedere, abbiamo ottenuto i migliori risultati dalla ricerca a griglia utilizzando il token inizializzatore standard, senza lo stemming di Porter, nessuna libreria di stop-word e tf-idfs in combinazione con un classificatore a regressione logistica con un'intensità di regolarizzazione L2 pari a `C=10.0`.

Utilizzando il modello migliore fornito dalla ricerca a griglia, stampiamo i punteggi di accuratezza con convalida incrociata a cinque parti sul set di addestramento e l'accuratezza della classificazione su un dataset di test:

```
>>> print('CV Accuracy: %.3f'
...       % gs_lr_tfidf.best_score_)
CV Accuracy: 0.897
>>> clf = gs_lr_tfidf.best_estimator_
>>> print('Test Accuracy: %.3f'
...     % clf.score(X_test, y_test))
Test Accuracy: 0.899
```

I risultati rivelano che il nostro modello di apprendimento è in grado di prevedere se una recensione di film è stata positiva o negativa con un'accuratezza del 90%.

NOTA

Un classificatore molto popolare per l'analisi del testo è Naïve Bayes, che ha acquisito popolarità nelle applicazioni di filtraggio dello spam nella posta elettronica. I classificatori Naïve Bayes sono facili da implementare, efficienti dal punto di vista computazionale e tendono a comportarsi particolarmente bene, rispetto ad altri algoritmi, su dataset relativamente compatti. Sebbene in questo testo non parleremo dei classificatori Naïve Bayes, il lettore interessato può trovare un articolo dell'autore sull'argomento, reso liberamente disponibile su arXiv (S. Raschka, *Naive Bayes and Text Classification I - introduction and Theory*, in "Computing Research Repository (CoRR)", abs/1410.5329, 2014. http://arxiv.org/pdf/1410.5329v3.pdf).

Lavorare su grossi insiemi di dati: algoritmi online e apprendimento out-of-core

Se avete eseguito gli esempi di codice del paragrafo precedente, avrete notato che potrebbe essere piuttosto costoso dal punto di vista computazionale costruire i vettori delle caratteristiche per il dataset delle 50.000 recensioni di film durante la ricerca a griglia. In molte applicazioni pratiche, può facilmente capitare di lavorare con dataset di dimensioni anche maggiori, in grado di superare perfino la memoria del computer. Poiché non tutti hanno accesso a potenze di calcolo da supercomputer, applicheremo una tecnica chiamata apprendimento *out-of-core*, che ci consente di lavorare con dataset così estesi.

Tornando al Capitolo 2, *I classificatori di machine learning di scikit-learn*, abbiamo introdotto il concetto di *pendenza del gradiente stocastica*, cioè un algoritmo di ottimizzazione che aggiorna i pesi del modello utilizzando un campione alla volta. In questo paragrafo faremo uso della funzione `partial_fit` di `SGDClassifier` in scikit-learn per ottenere i documenti direttamente dall'unità dischi locale e addestrare un modello a regressione logistica utilizzando piccole parti di documenti.

Innanzitutto, definiamo una funzione `tokenizer`, che ripulisce i dati in testuali grezzi del file `movie_data.csv` che abbiamo costruito all'inizio di questo capitolo e li separa in blocchi di parole, eliminando le stop word.

```
>>> import numpy as np
>>> import re
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> def tokenizer(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
...                            text.lower())
...     text = re.sub('[\W]+', ' ', text.lower()) \
...            + ' '.join(emoticons).replace('-', '')
...     tokenized = [w for w in text.split() if w not in stop]
...     return tokenized
```

Poi definiamo una funzione generatore, `stream_docs`, che legge e restituisce un documento alla volta:

```
>>> def stream_docs(path):
...     with open(path, 'r', encoding='utf-8') as csv:
...         next(csv) # skip header
...         for line in csv:
...             text, label = line[:-3], int(line[-2])
...             yield text, label
```

Per verificare che la funzione stream_docs funzioni correttamente, leggiamo il primo documento dal file movie_data.csv, che dovrebbe restituire una tupla costituita dal testo della recensione e dalla corrispondente etichetta della classe:

```
>>> next(stream_docs(path='./movie_data.csv'))
('"In 1974, the teenager Martha Moxley ... ',1)
```

Ora definiremo una funzione, get_minibatch, che prenderà il flusso di documenti dalla funzione stream_docs e restituirà un determinato numero di documenti, specificato dal parametro size:

```
>>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...         try:
...             for _ in range(size):
...                 text, label = next(doc_stream)
...                 docs.append(text)
...                 y.append(label)
...         except StopIteration:
...             return None, None
...         return docs, y
```

Sfortunatamente, non possiamo utilizzare CountVectorizer per l'apprendimento out-of-core, in quanto richiede la necessità di conservare l'intero vocabolario in memoria. Inoltre, TfidfVectorizer deve conservare tutti i vettori delle caratteristiche del dataset di addestramento in memoria per poter calcolare le frequenze inverse nel documento. Tuttavia, un altro utile vettorializzatore per l'elaborazione del testo implementato in scikit-learn si chiama HashingVectorizer, il quale è indipendente dai dati e utilizza la tecnica di hashing tramite l'algoritmo a 32-bit MurmurHash3 di Austin Appleby (https://sites.google.com/site/murmurhash/).

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> from sklearn.linear_model import SGDClassifier
>>> vect = HashingVectorizer(decode_error='ignore',
...                          n_features=2**21,
...                          preprocessor=None,
...                          tokenizer=tokenizer)
>>> clf = SGDClassifier(loss='log', random_state=1, n_iter=1)
>>> doc_stream = stream_docs(path='./movie_data.csv')
```

Utilizzando il codice precedente, abbiamo inizializzato HashingVectorizer con la nostra funzione tokenizer e abbiamo impostato il numero delle caratteristiche a 2^{21}. Inoltre, abbiamo inizializzato un classificatore a regressione logistica impostando il parametro loss del SGDClassifier a log. Notate che, scegliendo un numero elevato di caratteristiche in HashingVectorizer, riduciamo la probabilità di produrre collisioni hash, ma incrementiamo anche il numero di coefficienti nel modello a regressione logistica.

Ora arriva la parte davvero interessante. Avendo configurato tutte le funzioni complementari, possiamo ora iniziare l'apprendimento out-of-core utilizzando il codice seguente:

```
>>> import pyprind
>>> pbar = pyprind.ProgBar(45)
>>> classes = np.array([0, 1])
>>> for _ in range(45):
...     X_train, y_train = get_minibatch(doc_stream, size=1000)
...     if not X_train:
...         break
...     X_train = vect.transform(X_train)
...     clf.partial_fit(X_train, y_train, classes=classes)
...     pbar.update()
0%                      100%
[##############################] | ETA[sec]: 0.000
Total time elapsed: 50.063 sec
```

Di nuovo facciamo uso del package PyPrind per stimare il progresso dell'algoritmo di apprendimento. Abbiamo inizializzato un oggetto barra di progressione con 45 iterazioni e, nel successivo ciclo `for`, abbiamo iterato su 45 mini-lotti di documenti, dove ogni mini-lotto è costituito da 1000 documenti.

Dopo aver completato il processo di apprendimento incrementale, utilizzeremo gli ultimi 5000 documenti per valutare le prestazioni del modello:

```
>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
>>> X_test = vect.transform(X_test)
>>> print('Accuracy: %.3f' % clf.score(X_test, y_test))
Accuracy: 0.868
```

Come possiamo vedere, l'accuratezza del modello è pari all'87%, leggermente inferiore all'accuratezza che abbiamo ottenuto nel paragrafo precedente utilizzando la ricerca a griglia per l'ottimizzazione degli iperparametri. Tuttavia, l'apprendimento out-of-core è molto efficiente in termini di uso della memoria e la sua esecuzione ha richiesto meno di un minuto. Infine, possiamo utilizzare gli ultimi 5000 documenti per aggiornare il modello:

```
>>> clf = clf.partial_fit(X_test, y_test)
```

Se pensate di procedere direttamente al Capitolo 9, *Embedding di un modello in un'applicazione web*, vi consiglio di mantenere aperta l'attuale sessione di Python. Nel prossimo capitolo, utilizzeremo il modello che abbiamo appena addestrato per imparare a salvarlo su disco, in modo da riutilizzarlo e incorporarlo in un'applicazione web.

NOTA

Sebbene il modello bag-of-words sia tuttora quello più comunemente utilizzato per la classificazione del testo, non considera la struttura della frase e la grammatica. Una comune estensione del modello bag-of-words si chiama *allocazione Latent Dirichlet*, un particolare modello che considera la semantica latente delle parole (D. M. Blei, A. Y. Ng e M. I. Jordan, *Latent Dirichlet allocation*, in "The Journal of Machine Learning research", 3:993–1022, 2003).

Un'alternativa più moderna al modello bag-of-words è costituita da *word2vec*, un algoritmo che Google ha rilasciato nel 2013 (T. Mikolov, K. Chen, G. Corrado e J. Dean, *Efficient Estimation of Word Representations in Vector Space*, in "arXiv preprint arXiv":1301.3781, 2013). L'algoritmo word2vec è un algoritmo di apprendimento senza supervisione che si basa su reti neurali e che tenta di apprendere automaticamente le relazioni esistenti fra le parole. L'idea su

cui si basa word2vec consiste nel porre le parole che hanno significati simili in cluster simili; tramite un'intelligente spaziatura dei vettori, il modello può riprodurre determinate parole utilizzando un semplice operazione di matematica vettoriale, per esempio *king – man + woman = queen*. L'implementazione originale in linguaggio C, con link utili a documenti e implementazione alternative si trova in https://code.google.com/p/word2vec/.

Riepilogo

In questo capitolo abbiamo imparato a utilizzare gli algoritmi di machine learning per classificare documenti testuali sulla base della loro emotività, che è un compito di base nel campo dell'elaborazione del linguaggio naturale. Abbiamo imparato a codificare un documento come un vettore delle caratteristiche utilizzando il modello bag-of-words, e abbiamo imparato a pesare la frequenza dei termini per rilevanza, utilizzando la frequenza inversa.

Lavorare su dati testuali può essere piuttosto costoso dal punto di vista computazionale, a causa dei grandi vettori di caratteristiche che vengono creati durante l'elaborazione. Nell'ultimo paragrafo abbiamo imparato a utilizzare l'apprendimento out-of-core, o incrementale, per addestrare un algoritmo di machine learning senza dover caricare l'intero dataset nella memoria del computer.

Nel prossimo capitolo utilizzeremo il nostro classificatore di documenti e impareremo a incorporarlo in un'applicazione web.

Embedding di un modello in un'applicazione web

Nei capitoli precedenti abbiamo introdotto molti concetti legati alle attività di machine learning e presentato vari algoritmi che ci hanno aiutato a prendere decisioni in modo più efficiente. Tuttavia, le tecniche di machine learning non si limitano alle applicazioni e alle analisi offline e possono rappresentare un motore predizionale dei servizi web. Per esempio, fra le applicazioni più note e utili dei modelli di machine learning in ambito web vi sono il rilevamento dello spam nei moduli inoltrati, i motori di ricerca, i sistemi di suggerimenti per i portali multimediali o di vendita e molto altro ancora.

In questo capitolo impareremo a incorporare il modello di machine learning in un'applicazione web in grado non solo di classificare, ma anche di imparare dai dati in tempo reale. Questi sono gli argomenti trattati nel presente capitolo.

- Salvataggio dello stato corrente di un modello di machine learning addestrato.
- Uso di database SQLite per la memorizzazione dei dati.
- Sviluppo di un'applicazione web utilizzando il noto framework web Flask.
- Sviluppo di un'applicazione di machine learning su un server web pubblico.

Serializzazione di uno stimatore scikit-learn non addestrato

L'addestramento di un modello di machine learning può essere piuttosto costoso dal punto di vista computazionale, come abbiamo visto anche nel Capitolo 8, *Tecniche di machine learning per l'analisi del sentiment*. Sicuramente, non vogliamo addestrare il nostro modello ogni volta che chiudiamo il nostro interprete Python e dobbiamo effettuare una nuova previsione o ricaricare la nostra applicazione web! Un'opzione di persistenza del modello è costituita dal *modello persistence* di Python (https://docs.python.org/3.4/library/pickle.html), che ci consente di serializzare e deserializzare strutture di oggetti Python per compattare il bytecode, in modo da poter salvare lo stato corrente del nostro classificatore e ricaricarlo ogni volta che dobbiamo classificare nuovi campioni senza dover ogni volta addestrare il modello su nuovi dati. Prima di eseguire il codice seguente, assicuratevi però di aver addestrato il modello a regressione logistica *out-of-core* del capitolo precedente e di averlo quindi pronto nella sessione Python corrente:

```
>>> import pickle
>>> import os
>>> dest = os.path.join('movieclassifier', 'pkl_objects')
>>> if not os.path.exists(dest):
...     os.makedirs(dest)
>>> pickle.dump(stop,
...          open(os.path.join(dest, 'stopwords.pkl'),'wb'),
...          protocol=4)
>>> pickle.dump(clf,
...          open(os.path.join(dest, 'classifier.pkl'), 'wb'),
...          protocol=4)
```

Utilizzando il codice precedente, abbiamo creato una directory `movieclassifier`, nella quale conserveremo successivamente i file e i dati della nostra applicazione web. Nella directory `movieclassifier`, abbiamo creato una subdirectory `pkl_objects` per salvare sul disco rigido locale gli oggetti Python serializzati. Tramite il metodo di `dump` di `pickle`, abbiamo poi serializzato il modello a regressione logistica addestrato, così come il set di stop word dalla libreria NLTK, in modo da non dover installare il vocabolario NLTK sul nostro server. Il metodo `dump` accetta come primo argomento l'oggetto che vogliamo selezionare; come secondo argomento abbiamo fornito il file, aperto, sul quale l'oggetto Python verrà scritto. Tramite l'argomento `wb` all'interno della funzione `open`, abbiamo aperto il file in modalità binaria per l'estrazione e abbiamo impostato `protocol=4` per scegliere l'ultimo e più efficiente protocollo di `pickle` che è stato aggiunto a Python 3.4. Se avete problemi a utilizzare il protocollo 4, verificate di

aver installato l'ultima versione di Python 3. Alternativamente, potete scegliere un protocollo inferiore.

> **NOTA**
> Il nostro modello a regressione logistica contiene numerosi array NumPy, come il vettore dei pesi, e un modo più efficiente per serializzare gli array NumPy consiste nell'utilizzare la libreria alternativa joblib. Per garantire la compatibilità con l'ambiente server che utilizzeremo nei prossimi paragrafi, utilizzeremo l'approccio pickle standard. Se siete interessati, potete trovare ulteriori informazioni su joblib in https://pypi.python.org/pypi/joblib.

Non abbiamo bisogno di applicare pickle al HashingVectorizer, poiché non ha alcuna necessità di essere adattato. Piuttosto, possiamo creare un nuovo file scritto in Python, dal quale possiamo importare i vettorizzatori nella nostra sessione Python corrente. Ora, copiate il codice seguente e salvatelo come vectorizer.py nella directory movieclassifier:

```
from sklearn.feature_extraction.text import HashingVectorizer
import re
import os
import pickle

cur_dir = os.path.dirname(__file__)
stop = pickle.load(open(
            os.path.join(cur_dir,
            'pkl_objects',
            'stopwords.pkl'), 'rb'))

def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
                           text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) \
                + ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized

vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)
```

Dopo aver salvato gli oggetti Python e aver creato il file vectorizer.py, sarebbe una buona idea riavviare l'interprete Python o il kernel IPython Notebook per verificare se siamo in grado di deserializzare gli oggetti senza errori. Tuttavia, notate che il successivo caricamento dei dati da una fonte non fidata può rappresentare un potenziale rischio per la sicurezza, poiché il modulo pickle non è protetto contro il codice pericoloso. Dal terminale, raggiungete la directory movieclassifier, avviate una nuova sessione di Python ed eseguite il codice seguente, per verificare di poter importare il vettorizzatore e caricare il classificatore:

```
>>> import pickle
>>> import re
>>> import os
>>> from vectorizer import vect
>>> clf = pickle.load(open(
...       os.path.join('pkl_objects',
...                    'classifier.pkl'), 'rb'))
```

Dopo aver caricato con successo il vettorizzatore e il classificatore, possiamo utilizzare questi oggetti per pre-elaborare i campioni di documenti ed eseguire previsioni sul sentiment:

```
>>> import numpy as np
>>> label = {0:'negative', 1:'positive'}
>>> example = ['I love this movie']
>>> X = vect.transform(example)
>>> print('Prediction: %s\nProbability: %.2f%%' %\
...       (label[clf.predict(X)[0]],
...        np.max(clf.predict_proba(X))*100))
Prediction: positive
Probability: 91.56%
```

Poiché il nostro classificatore restituisce le etichette delle classi sotto forma di interi, abbiamo definito un semplice dizionario Python per mappare tali interi sul sentiment. Poi abbiamo utilizzato `HashingVectorizer` per trasformare il semplice documento d'esempio in un vettore di parole `X`. Infine abbiamo utilizzato il metodo `predict` del classificatore a regressione logistica per prevedere l'etichetta della classe e il metodo `predict_proba` per restituire la corrispondente probabilità della previsione. Notate che il metodo `predict_proba` restituisce un array con un valore di probabilità per ogni etichetta univoca di classe. Poiché l'etichetta della classe con le probabilità maggiori corrisponde all'etichetta della classe restituita dalla chiamata a `predict`, abbiamo utilizzato la funzione `np.max` per restituire la probabilità della classe prevista.

Impostazione di un database SQLite per la memorizzazione dei dati

In questo paragrafo configureremo un semplice *database SQLite* per raccogliere dei feedback opzionali dagli utenti dell'applicazione web. Possiamo utilizzare questo feedback per aggiornare il nostro modello di classificazione. SQLite è un engine open source per database SQL che non richiede un server esterno per poter funzionare; questo lo rende una soluzione ideale per piccoli progetti e semplici applicazioni web. Sostanzialmente, un database SQLite è costituito da un unico file, il quale ci consente di accedere direttamente ai file contenenti i dati. Inoltre, SQLite non richiede alcuna configurazione specifica ed è supportato da tutti i sistemi operativi più comuni. Si è guadagnato la sua reputazione per la sua affidabilità e ora è utilizzato dalle più note aziende, come Google, Mozilla, Adobe, Apple, Microsoft e molte altre ancora. Per informazioni su SQLite, potete consultare il suo sito web all'indirizzo http://www.sqlite.org.

Fortunatamente, Python, seguendo la sua filosofia "batterie incluse", offre già un'API nella sua libreria standard, *sqlite3*, che ci consente di lavorare con i database SQLite (per informazioni su sqlite3, visitate l'indirizzo https://docs.python.org/3.4/library/sqlite3.html).

Tramite il codice seguente, creeremo un nuovo database SQLite all'interno della directory `movieclassifier` e vi memorizzeremo due esempi di recensioni di film:

```
>>> import sqlite3
>>> import os
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute('CREATE TABLE review_db'\
...           ' (review TEXT, sentiment INTEGER, date TEXT)')
>>> example1 = 'I love this movie'
>>> c.execute("INSERT INTO review_db"\
...           " (review, sentiment, date) VALUES"\
...           " (?, ?, DATETIME('now'))", (example1, 1))
>>> example2 = 'I disliked this movie'
>>> c.execute("INSERT INTO review_db"\
...           " (review, sentiment, date) VALUES"\
...           " (?, ?, DATETIME('now'))", (example2, 0))
>>> conn.commit()
>>> conn.close()
```

Dopo il codice precedente, abbiamo creato una connessione (`conn`) con un database SQLite richiamando il metodo `connect` di `sqlite3`, che crea il nuovo file `reviews.sqlite` nella directory `movieclassifier`, qualora tale file non esista. Notate che SQLite non implementa una funzione di sostituzione di eventuali tabelle esistenti; sarà pertanto necessario cancellare manualmente il file del database per eseguire il codice una seconda volta.

Poi abbiamo creato un cursore utilizzando il metodo cursor, che ci consente di attraversare i record del database sfruttando la potente sintassi SQL. Tramite la prima chiamata a execute abbiamo creato una nuova tabella del database, review_db. L'abbiamo utilizzata per accedere ai record del database. Insieme a review_db, abbiamo creato anche tre colonne nella tabella del database: review, sentiment e date. Le abbiamo utilizzate per conservare due recensioni d'esempio e le rispettive etichette delle classi (sentimenti). Utilizzando il comando SQL DATETIME('now'), abbiamo inoltre aggiunto alle voci le indicazioni temporali sulla loro creazione (*timestamp*). Oltre ai timestamp, abbiamo utilizzato il simbolo del punto interrogativo (?) per passare come argomenti posizionali al metodo execute i testi delle recensioni sui film (example1 e example2) e le etichette delle classi corrispondenti (1 e 0) quali membri di una tupla. Infine abbiamo chiamato il metodo commit per salvare le modifiche che abbiamo apportato al database e abbiamo chiuso la connessione tramite il metodo close.

Per controllare se le voci sono state memorizzate correttamente nella tabella del database, riapriremo la connessione al database e utilizzeremo il comando SQL SELECT per leggere tutte le righe dalla tabella del database che sono state redatte fra l'inizio del 2015 a oggi:

```
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute("SELECT * FROM review_db WHERE date"\
...     " BETWEEN '2015-01-01 00:00:00' AND DATETIME('now')")
>>> results = c.fetchall()
>>> conn.close()
>>> print(results)
[('I love this movie', 1, '2015-06-02 16:02:12'),
 ('I disliked this movie', 0, '2015-06-02 16:02:12')]
```

Alternativamente, avremmo potuto utilizzare il plugin gratuito del browser Firefox *SQLite Manager* (disponibile presso https://addons.mozilla.org/it/firefox/addon/sqlite-manager/), che offre un'interessante interfaccia grafica per lavorare con i database SQLite, come si può vedere dalla Figura 9.1.

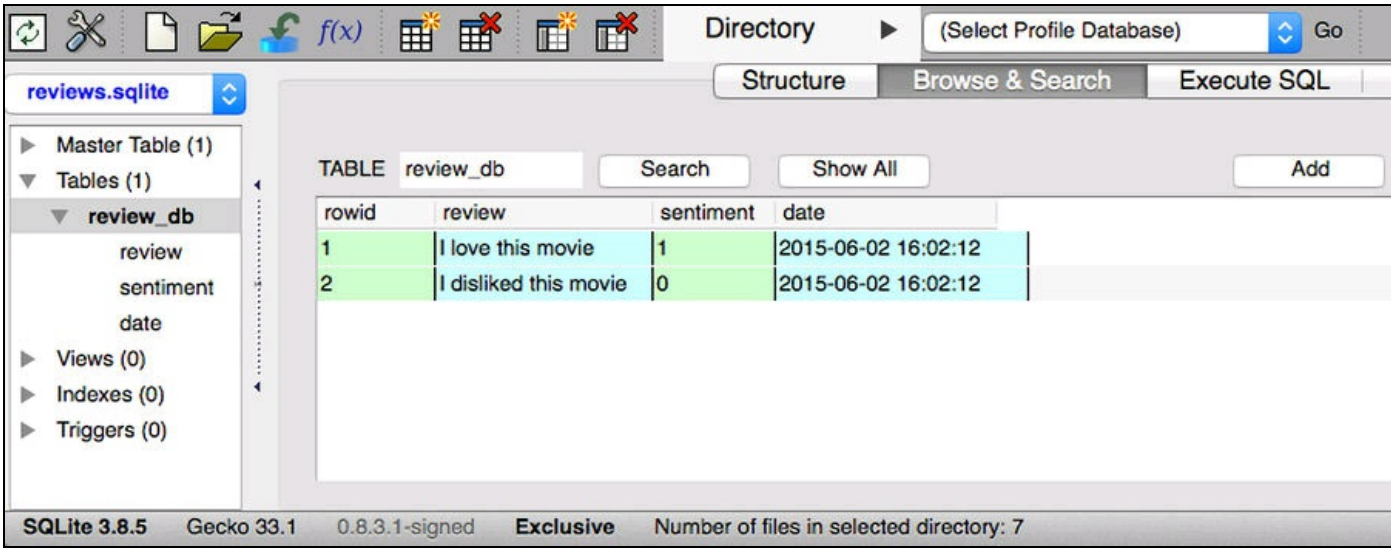

Directory
(Select Profile Database)
Go
reviews.sqlite
Structure
Browse & Search
Execute SQL
Master Table (1)
Tables (1)
review_db
review
sentiment
date
Views (0)
Indexes (0)
Triggers (0)
TABLE review_db
Search
Show All
Add
rowid
review
sentiment
date
1
I love this movie
1
2015-06-02 16:02:12
2
I disliked this movie
0
2015-06-02 16:02:12
SQLite 3.8.5
Gecko 33.1
0.8.3.1-signed
Exclusive
Number of files in selected directory: 7

Figura 9.1

Sviluppare un'applicazione web con Flask

Dopo aver preparato il codice per classificare le recensioni sui film, parliamo delle basi del framework web Flask per lo sviluppo di applicazioni web. Dopo che Armin Ronacher ha prodotto la release iniziale di Flask, nel 2010, il framework ha acquisito grande popolarità nel corso degli anni e fra gli esempi di applicazioni che fanno uso di Flask vi sono LinkedIn e Pinterest. Poiché Flask è scritto in Python, ci fornisce, come programmatori, una comoda interfaccia per l'inclusione del codice Python esistente, come nel caso del classificatore dei film.

NOTA

Flask è chiamato anche *micro-framework*, in quanto il suo core viene mantenuto intenzionalmente leggero e semplice, ma può essere esteso con facilità con altre librerie. Sebbene la curva di apprendimento dell'API di Flask non sia ripida quanto quella di altri framework web per Python, come Django, si consiglia comunque di dare un'occhiata alla documentazione ufficiale di Flask all'indirizzo http://flask.pocoo.org/docs/0.10/, per imparare qualcosa di più sulle funzionalità da esso offerte.

Se la libreria Flask non è già installata nel vostro attuale ambiente Python, potete installarla con facilità tramite `pip` dal Terminale (al momento attuale, l'ultima versione stabile era la 0.10.1):

```
pip install flask
```

Una prima applicazione web con Flask

In questo paragrafo svilupperemo una semplicissima applicazione web per acquisire familiarità con l'API di Flask, prima di implementare il classificatore dei film. Innanzitutto predisponiamo un albero di directory:

```
1st_flask_app_1/
  app.py
  templates/
    first_app.html
```

Il file `app.py` conterrà il codice principale che verrà eseguito dall'interprete Python per impiegare l'applicazione web Flask. La directory `templates` è quella in cui Flask ricercherà i file HTML statici per la visualizzazione nel browser web. Diamo un'occhiata al contenuto del file `app.py`:

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
  return render_template('first_app.html')
```

```
if __name__ == '__main__':
    app.run()
```

In questo caso, eseguiamo l'applicazione come un unico modulo, pertanto dobbiamo inizializzare una nuova istanza di Flask con l'argomento `__name__` per far sapere a Flask che può trovare la cartella template HTML (`templates`) nella stessa directory in cui è situato. Poi utilizziamo il decoratore (`@app.route('/')`) per specificare l'URL che dovrebbe lanciare l'esecuzione della funzione `index`. Qui, la nostra funzione `index` non fa altro che produrre il file HTML `first_app.html`, situato nella cartella `templates`. Infine, utilizziamo la funzione `run` per eseguire l'applicazione sul server solo quando questo script viene eseguito direttamente dall'interprete Python, cosa che abbiamo garantito utilizzando l'istruzione `if` con `__name__ == '__main__'`.

Ora diamo un'occhiata al contenuto del file `first_app.html`. Se non siete pratici della sintassi HTML, può essere utile una visita al sito http://www.w3schools.com/html/default.asp, dove troverete guide utili per l'apprendimento delle basi.

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  </head>
  <body>
  <div>Hi, this is my first Flask web app!</div>
  </body>
</html>
```

Qui abbiamo semplicemente riempito un file template HTML con un elemento `div` (un elemento a livello di blocco) che contiene la frase `Hi, this is my first Flask web app!`. Opportunamente, Flask ci consente di eseguire le nostre applicazioni localmente, cosa che può essere utile in fase di sviluppo, per collaudare le applicazioni web prima di inviarle sul server web pubblico. Ora, avviamo l'applicazione web eseguendo il comando dal terminale all'interno della directory `1st_flask_app_1`:

```
python3 app.py
```

Dovremmo ottenere una riga simile alla seguente:

```
* Running on http://127.0.0.1:5000/
```

Questa riga contiene l'indirizzo del nostro server locale. Ora possiamo inserire questo indirizzo nel nostro browser web per vedere l'applicazione in funzione. Se tutto viene eseguito correttamente, dovremmo trovare un semplice sito web con il seguente contenuto: **Hi, this is my first Flask web app!**.

Convalida e rendering del modulo

Estenderemo la nostra semplice applicazione web in Flask con elementi per moduli HTML per imparare a raccogliere dati dall'utente utilizzando la libreria *WTForms* (https://wtforms.readthedocs.org/en/latest/), che può essere installata tramite pip:

```
pip install wtforms
```

Questa applicazione web chiede all'utente di digitare il proprio nome in un campo testuale, come si può vedere nella Figura 9.2.

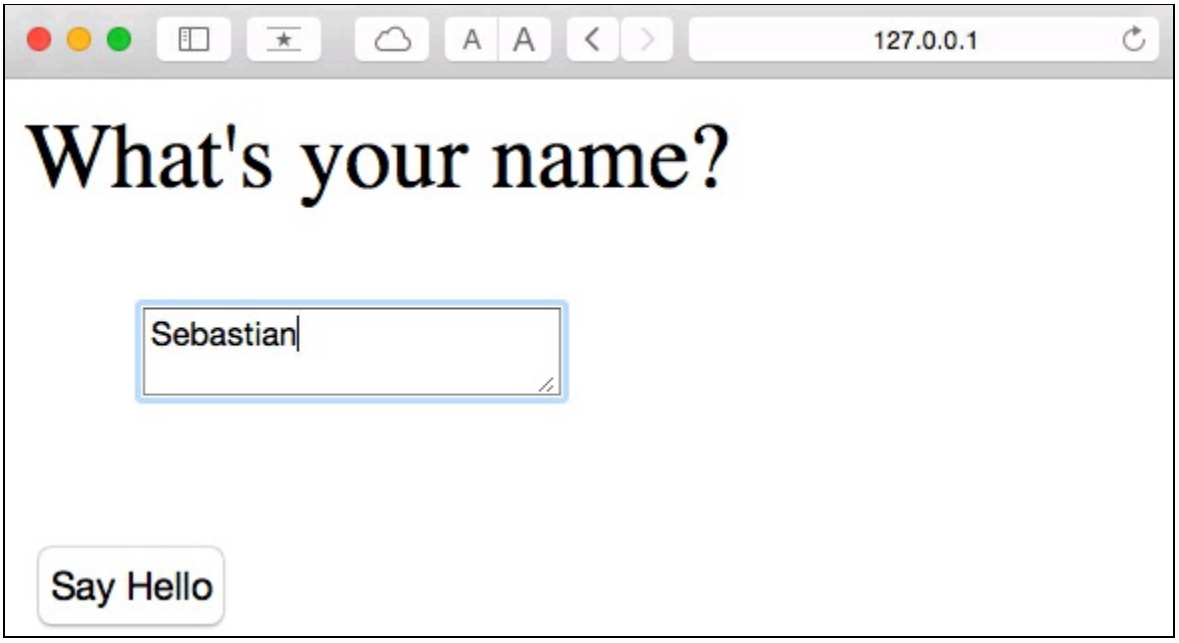


Figura 9.2

Dopo aver fatto clic sul pulsante *Say Hello* e dopo la convalida del modulo, verrà prodotta una nuova pagina HTML per visualizzare il nome dell'utente (Figura 9.3).

Figura 9.3

La nuova struttura di directory di cui abbiamo bisogno per configurare questa applicazione avrà il seguente aspetto:

```
1st_flask_app_2/
  app.py
  static/
    style.css
  templates/
    _formhelpers.html
    first_app.html
    hello.html
```

Quello che segue è il contenuto modificato del file app.py:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators

app = Flask(__name__)

class HelloForm(Form):
    sayhello = TextAreaField('',[validators.DataRequired()])

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)

if __name__ == '__main__':
    app.run(debug=True)
```

Utilizzando wtforms, abbiamo esteso la funzione index con un campo di testo che incorporeremo nella nostra pagina iniziale utilizzando la classe TextAreaField, la quale controlla automaticamente se un utente ha fornito o meno un input testuale valido. Inoltre, abbiamo definito una nuova funzione, hello, che produrrà una pagina HTML hello.html se il modulo viene convalidato. Qui abbiamo utilizzato il metodo POST per trasportare i dati del modulo al server, nel corpo del messaggio. Infine, impostando l'argomento debug=True all'interno del metodo app.run, abbiamo anche attivato il debugging Flask. Questa è una funzionalità utile per lo sviluppo di nuove applicazioni web.

Ora implementeremo una macro generica nel file _formhelpers.html tramite l'engine per template *Jinja2*, che successivamente importeremo nel nostro file first_app.html per produrre il campo di testo:

```
{% macro render_field(field) %}
 <dt>{{ field.label }}
 <dd>{{ field(**kwargs)|safe }}
 {% if field.errors %}
  <ul class=errors>
  {% for error in field.errors %}
   <li>{{ error }}</li>
  {% endfor %}
  </ul>
 {% endif %}
 </dd>
 </dt>
{% endmacro %}
```

Una discussione approfondita sul linguaggio per template Jinja2 non rientra negli scopi di questo libro. Tuttavia, potete trovare un'ampia documentazione della sintassi di Jinja2 all'indirizzo http://jinja.pocoo.org.

Ora configuriamo un semplice file *CSS* (*Cascading Style Sheets*), style.css, per illustrare come sia possibile modificare l'aspetto dei documenti HTML. Dobbiamo

salvare il seguente file CSS, che non fa altro che raddoppiare le dimensioni dei caratteri del corpo del testo HTML, in una subdirectory chiamata static, che è la directory standard nella quale Flask ricerca i file statici come i fogli stile CSS. Il codice è il seguente:

```
body {
  font-size: 2em;
}
```

Quello che segue è il contenuto del file first_app.html modificato, che a questo punto produce un modulo di testo che permette all'utente di specificare il proprio nome:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  <link rel="stylesheet" href="{{ url_for('static',
    filename='style.css') }}">
  </head>
  <body>

{% from "_formhelpers.html" import render_field %}

<div>What's your name?</div>
<form method=post action="/hello">
  <dl>
    {{ render_field(form.sayhello) }}
  </dl>
  <input type=submit value='Say Hello' name='submit_btn'>
</form>
  </body>
</html>
```

Nella sezione header del file first_app.html abbiamo caricato il file CSS. A questo punto dovrebbe modificare le dimensioni di tutti gli elementi testuali del corpo HTML. Nella sezione del corpo HTML, abbiamo importato la macro del modulo da _formhelpers.html e abbiamo prodotto il modulo sayhello che abbiamo specificato nel file app.py. Inoltre abbiamo aggiunto un pulsante, in modo che l'utente possa inviare quanto ha specificato nel campo di testo.

Infine, abbiamo creato un file hello.html che verrà prodotto tramite la riga render_template('hello.html', name=name) all'interno della funzione hello che abbiamo definito nello script app.py e che visualizzerà il testo che l'utente ha specificato nel campo. Il codice è il seguente:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  <link rel="stylesheet" href="{{ url_for('static',
    filename='style.css') }}">
  </head>
  <body>

<div>Hello {{ name }}</div>
  </body>
</html>
```

Dopo aver configurato l'applicazione web Flask modificata, possiamo eseguirla localmente lanciando il seguente comando dalla directory principale dell'app; possiamo vedere il risultato nel browser web all'indirizzo http://127.0.0.1:5000/:

```
python3 app.py
```

NOTA

Se non siete particolarmente ferrati nello sviluppo web, alcuni di questi concetti potrebbero sembrarvi oscuri. In tal caso, potete configurare i file precedenti in una directory del disco rigido e poi esaminarli attentamente. Noterete come il framework web Flask sia effettivamente molto semplice, molto di più di quanto possa apparire a prima vista! Inoltre, per ulteriori informazioni, non dimenticate di consultare l'eccellente documentazione relativa a Flask, ricca di esempi all'indirizzo http://flask.pocoo.org/docs/0.10/.

Trasformazione del classificatore di film in un'applicazione web

Ora che abbiamo appreso quanto meno le basi dello sviluppo web in Flask, procediamo con il passo successivo e implementiamo il nostro classificatore di film sotto forma di applicazione web. In questo paragrafo svilupperemo un'applicazione web che innanzitutto chiede all'utente di specificare una recensione, come illustrato nella Figura 9.4.

Dopo aver inviato la recensione, l'utente vedrà una nuova pagina, che mostra l'etichetta della classe prevista e la probabilità di tale previsione. Inoltre, l'utente potrà fornire il suo feedback su questa previsione, facendo clic sui pulsanti *Correct* o *Incorrect*, visibili nella Figura 9.5.

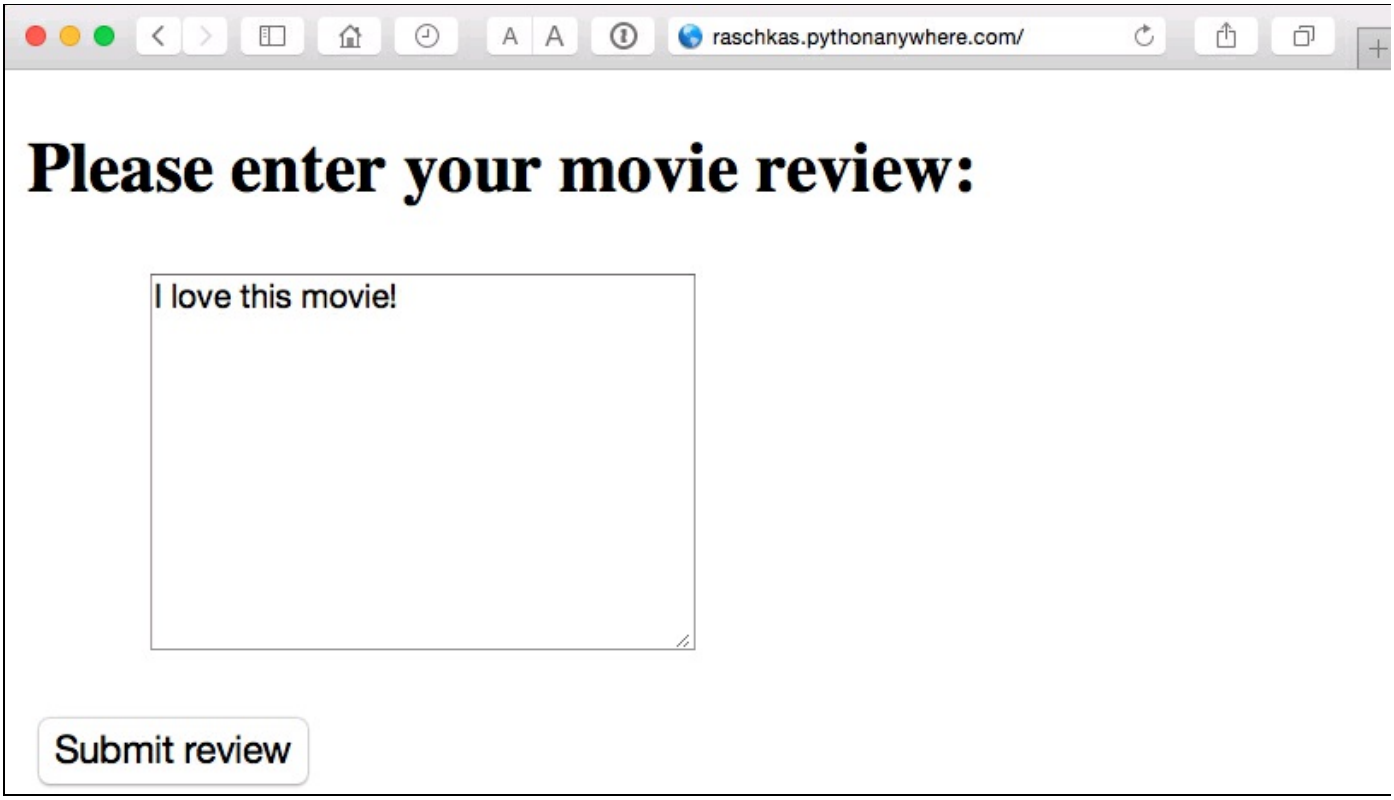


Figura 9.4

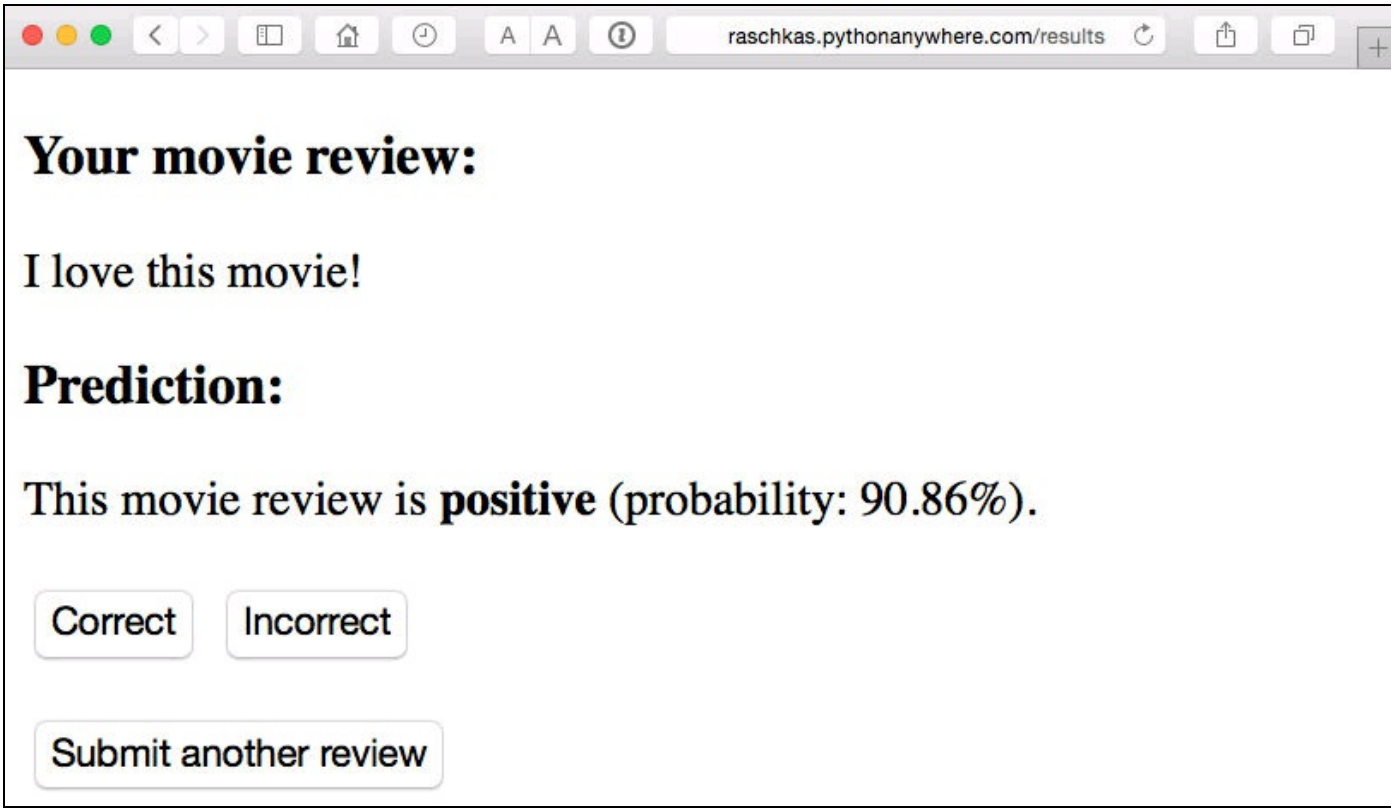


Figura 9.5

Se un utente ha fatto clic sul pulsante *Correct* o *Incorrect*, il modello di classificazione viene aggiornato considerando il feedback dell'utente. Inoltre, conserveremo in un database SQLite, per ogni riferimento futuro, il testo della recensione fornito dall'utente, come pure l'etichetta della classe suggerita, determinata sulla base del clic sul pulsante. La terza pagina che verrà vista dall'utente dopo aver fatto clic su uno dei due pulsanti di feedback è la semplice pagina di ringraziamento contenente il pulsante *Submit another review* che redirige l'utente di nuovo alla pagina iniziale (Figura 9.6).

Prima di consultare l'implementazione del codice per questa applicazione web, siete incoraggiati a dare un'occhiata alla demo live presente in http://raschkas.pythonanywhere.com, per comprendere meglio ciò che stiamo tentando di realizzare in questo paragrafo.

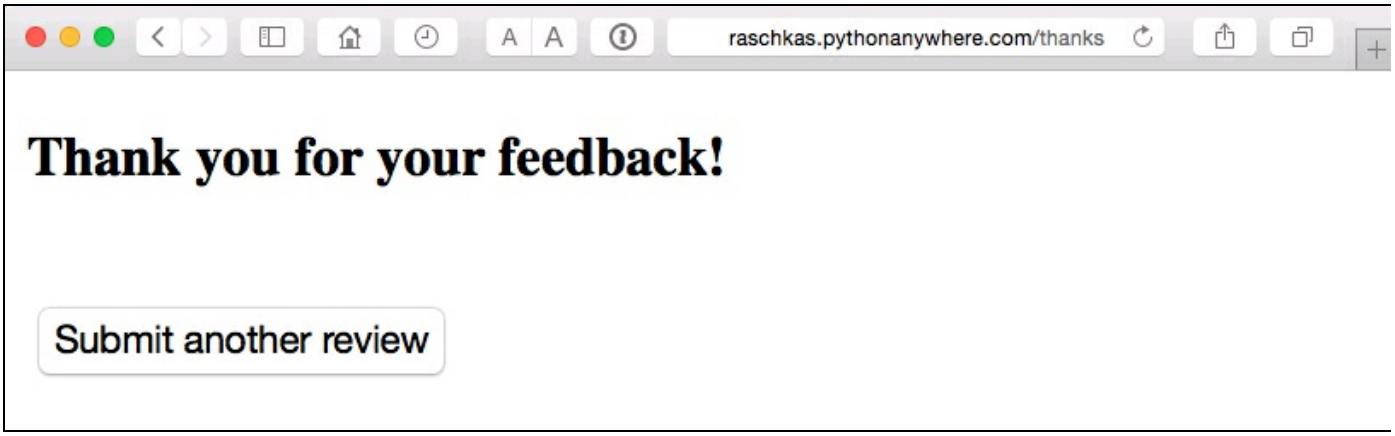


Figura 9.6

Per iniziare, diamo un'occhiata all'albero di directory che dobbiamo creare per questa applicazione per la classificazione di film (Figura 9.7).

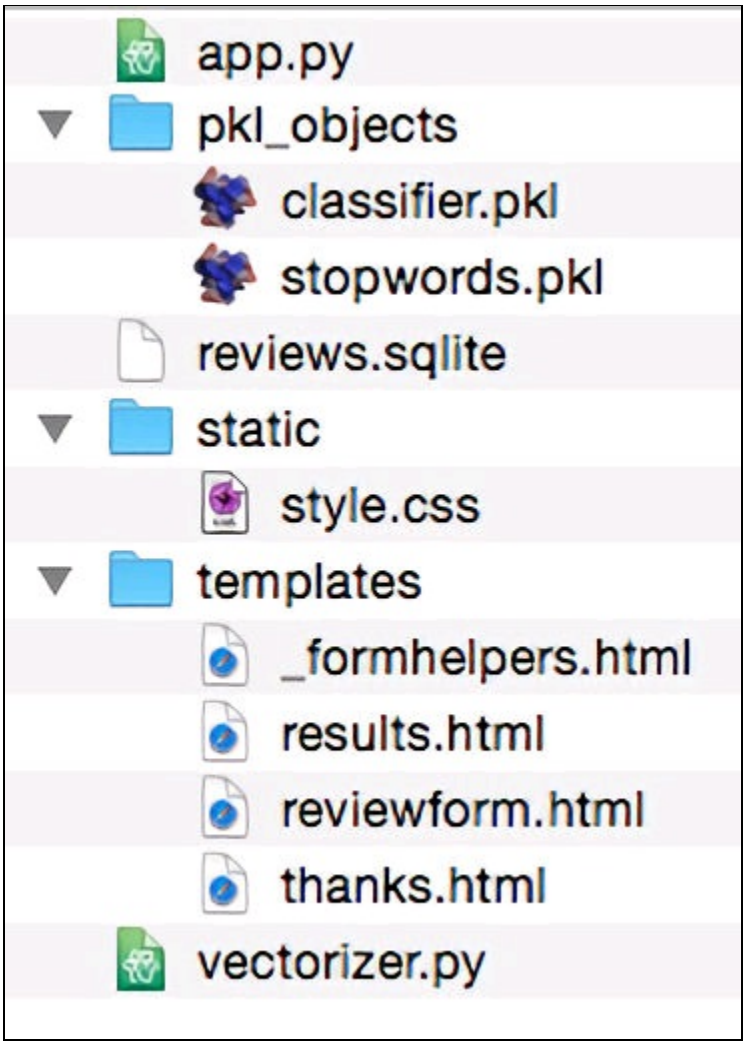


Figura 9.7

Nel paragrafo precedente di questo capitolo, abbiamo già creato il file `vectorizer.py`, il database SQLite `reviews.sqlite` e la subdirectory `pkl_objects` con gli oggetti Python.

Il file app.py nella directory principale è lo script Python che contiene il codice Flask e utilizzerà il file del database review.sqlite (che abbiamo creato in precedenza in questo capitolo) per conservare le recensioni dei film che verranno inviate all'applicazione web. La subdirectory templates contiene i template HTML che verranno mostrati da Flask nel browser e la subdirectory static conterrà un semplice file CSS per intervenire sull'aspetto del codice HTML prodotto.

Poiché il file app.py è piuttosto lungo, lo affronteremo in due passi. La prima parte del file app.py importa i moduli e gli oggetti Python di cui avremo bisogno, e anche il codice per caricare e configurare il modello di classificazione:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
import pickle
import sqlite3
import os
import numpy as np
# import HashingVectorizer from local dir
from vectorizer import vect

app = Flask(__name__)

######## Preparing the Classifier
cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                'pkl_objects/classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

def classify(document):
    label = {0: 'negative', 1: 'positive'}
    X = vect.transform([document])
    y = clf.predict(X)[0]
    proba =  clf.predict_proba(X).max()
    return label[y], proba

def train(document, y):
    X = vect.transform([document])
    clf.partial_fit(X, [y])

def sqlite_entry(path, document, y):
    conn = sqlite3.connect(path)
    c = conn.cursor()
    c.execute("INSERT INTO review_db (review, sentiment, date)"\
    " VALUES (?, ?, DATETIME('now'))", (document, y))
    conn.commit()
    conn.close()
```

Questa prima parte dello script app.py dovrebbe risultare piuttosto familiare. Non facciamo altro che importare HashingVectorizer e caricare il classificatore a regressione logistica. La funzione classify restituisce l'etichetta della classe prevista e anche la probabilità della previsione di un determinato documento di testo. La funzione train può essere utilizzata per aggiornare il classificatore sulla base di un documento e di un'etichetta della classe. Utilizzando la funzione sqlite_entry possiamo memorizzare una recensione nel database SQLite insieme alla sua etichetta della classe e al suo timestamp, come registrazione personale. Notate che, se riavviamo l'applicazione

web. l'oggetto clf verrà riportato al suo stato originale. Alla fine di questo capitolo, vedremo come utilizzare i dati che abbiamo raccolto nel database SQLite per aggiornare in modo permanente il classificatore.

I concetti della seconda parte dello script app.py dovrebbero comunque risultare piuttosto familiari:

```
app = Flask(__name__)
class ReviewForm(Form):
    moviereview = TextAreaField('',
                        [validators.DataRequired(),
                        validators.length(min=15)])

@app.route('/')
def index():
    form = ReviewForm(request.form)
    return render_template('reviewform.html', form=form)

@app.route('/results', methods=['POST'])
def results():
    form = ReviewForm(request.form)
    if request.method == 'POST' and form.validate():
        review = request.form['moviereview']
        y, proba = classify(review)
        return render_template('results.html',
                            content=review,
                            prediction=y,
                            probability=round(proba*100, 2))
    return render_template('reviewform.html', form=form)

@app.route('/thanks', methods=['POST'])
def feedback():
    feedback = request.form['feedback_button']
    review = request.form['review']
    prediction = request.form['prediction']

    inv_label = {'negative': 0, 'positive': 1}
    y = inv_label[prediction]
    if feedback == 'Incorrect':
        y = int(not(y))
    train(review, y)
    sqlite_entry(db, review, y)
    return render_template('thanks.html')

if __name__ == '__main__':
    app.run(debug=True)
```

Abbiamo definito una classe ReviewForm che istanzia un campo TextAreaField, che verrà poi prodotto sullo schermo dal template reviewform.html (la pagina di arrivo della nostra applicazione web). Questa, a sua volta, viene prodotta sullo schermo tramite la funzione index. Con il parametro validators.length(min=15), richiediamo all'utente di inserire una recensione che contenga almeno quindici caratteri. All'interno della funzione results, leggiamo il contenuto del modulo web inoltrato e lo passiamo al classificatore perché individui il sentiment, il quale verrà poi visualizzato nel template results.html prodotto.

La funzione feedback può sembrare un po' più complicata, a prima vista. Sostanzialmente legge l'etichetta della classe prevista dal template results.html se un utente ha fatto clic sul pulsante di feedback *Correct* o *Incorrect* e trasforma il

sentiment previsto in un'etichetta della classe (un intero) che verrà utilizzata per aggiornare il classificatore tramite la funzione train, che abbiamo implementato nella prima parte dello script app.py. Inoltre, verrà creata una nuova voce nel database SQLite tramite la funzione sqlite_entry qualora venga fornito un feedback e alla fine verrà prodotto il template thanks.html per ringraziare l'utente del feedback.

Ora diamo un'occhiata al template reviewform.html, che costituisce la pagina iniziale dell'applicazione:

```
<!doctype html>
<html>
<head>
   <title>Movie Classification</title>
</head>
 <body>

<h2>Please enter your movie review:</h2>

{% from "_formhelpers.html" import render_field %}

<form method=post action="/results">
 <dl>
   {{ render_field(form.moviereview, cols='30', rows='10') }}
 </dl>
 <div>
   <input type=submit value='Submit review' name='submit_btn'>
 </div>
</form>

 </body>
</html>
```

Qui, abbiamo semplicemente importato lo stesso template _formhelpers.html che abbiamo definito nel paragrafo *Convalida e rendering del modulo*, presentato nelle pagine precedenti di questo capitolo. La funzione render_field di questa macro viene utilizzata per produrre un campo TextAreaField dove un utente può fornire la recensione del film che poi invierà tramite il pulsante *Submit review* visualizzato nella parte inferiore della pagina. Questo campo TextAreaField è costituito da 30 colonne e 10 righe.

Il nostro template successivo, results.html, ha un aspetto un po' più interessante:

```
<!doctype html>
<html>
 <head>
   <title>Movie Classification</title>
 <link rel="stylesheet" href="{{ url_for('static',
   filename='style.css') }}">
 </head>
 <body>

<h3>Your movie review:</h3>
<div>{{ content }}</div>

<h3>Prediction:</h3>
<div>This movie review is <strong>{{ prediction }}</strong>
 (probability: {{ probability }}%).</div>

<div class='button'>
 <form action="/thanks" method="post">
   <input type=submit value='Correct' name='feedback_button'>
   <input type=submit value='Incorrect' name='feedback_button'>
   <input type=hidden value='{{ prediction }}' name='prediction'>
```

```
    <input type=hidden value='{{ content }}' name='review'>
  </form>
</div>

<div class='button'>
  <form action="/">
    <input type=submit value='Submit another review'>
  </form>
</div>

  </body>
</html>
```

Innanzitutto abbiamo inserito la recensione inviata e anche i risultati della previsione nei campi corrispondenti (`{{ content }}`, `{{ prediction }}` e `{{ probability }}`). Potete notare che abbiamo utilizzato le variabili segnaposto `{{ content }}` e `{{ prediction }}` una seconda volta nel modulo che contiene i pulsanti *Correct* e *Incorrect*. Questo è un trucco per inviare con `POST` questi valori di nuovo al server, per aggiornare il classificatore e per memorizzare la recensione qualora l'utente abbia fatto clic su uno di questi due pulsanti. Inoltre, abbiamo importato un file CSS (`style.css`) all'inizio del file `results.html`. La configurazione di questo file è piuttosto semplice: limita la larghezza del contenuto di questa applicazione web a 600 pixel e sposta verso il basso di 20 pixel i pulsanti *Incorrect* e *Correct*, etichettati con `button`:

```
body{
  width:600px;
}
.button{
  padding-top: 20px;
}
```

Questo file CSS è solo un segnaposto, che potete personalizzare per controllare l'aspetto e il comportamento dell'applicazione web.

L'ultimo file HTML che implementeremo per l'applicazione web è relativo al template `thanks.html`. Come suggerisce il nome, presenta semplicemente un messaggio di ringraziamento all'utente dopo che ha fornito il suo feedback tramite il pulsante *Correct* o *Incorrect*. Inoltre, inseriamo un pulsante *Submit another review* nella parte inferiore di questa pagina, che redirigerà l'utente sulla pagina iniziale. Il contenuto del file `thanks.html` è il seguente:

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
</head>
  <body>

<h3>Thank you for your feedback!</h3>
<div id='button'>
  <form action="/">
    <input type=submit value='Submit another review'>
  </form>
</div>

  </body>
</html>
```

Ora sarebbe una buona idea avviare l'applicazione web localmente dal terminale, tramite il seguente comando prima di procedere con il paragrafo successivo nel quale pubblicheremo l'applicazione su un server web pubblico:

```
python3 app.py
```

Dopo aver eseguito il collaudo dell'applicazione, non dovremmo dimenticarci di eliminare l'argomento `debug=True` nel comando `app.run()` del nostro script `app.py`.

Pubblicazione dell'applicazione web su un server pubblico

Dopo aver collaudato localmente l'applicazione web, siamo pronti per inviarla a un server pubblico. Per questo esempio, utilizzeremo il servizio di hosting web *PythonAnywhere*, specializzato nell'hosting di applicazioni web Python ed estremamente semplice ed esente da problemi. Inoltre, PythonAnywhere offre un account rivolto ai principianti, che consente di gestire un'unica applicazione web in modo completamente gratuito.

Per creare un nuovo account PythonAnywhere, visitate il sito web all'indirizzo https://www.pythonanywhere.com e fate clic sul link *Pricing & signup*, in alto a destra. Poi fate clic sul pulsante *Create a Beginner account* e fornite il nome utente, la password e un indirizzo di posta elettronica valido. Dopo aver letto e accettato i termini e le condizioni d'uso, potrete entrare in possesso del vostro nuovo account.

Sfortunatamente, l'account gratuito per principianti non consente di accedere al server remoto tramite il protocollo SSH dal Terminale della riga di comando. Pertanto dovrete utilizzare l'interfaccia web PythonAnywhere per gestire la vostra applicazione web. Prima di poter inviare sul server i file della vostra applicazione locale, dovete creare una nuova applicazione web per il vostro account PythonAnywhere. Dopo aver fatto clic sul pulsante Dashboard nell'angolo superiore destro, dovete accedere al pannello rappresentato nella parte superiore della pagina. Poi fate clic sulla scheda Web che risulta visibile nella parte superiore della pagina. Procedete facendo clic sul pulsante *Add a new web app*, a sinistra, che consente di creare una nuova applicazione web Python 3.4 Flask che potrete chiamare movieclassifier.

Dopo aver creato una nuova applicazione web per il vostro account PythonAnywhere, richiamate la scheda *Files* per inviare i file dalla directory locale movieclassifier utilizzando l'interfaccia web di PythonAnywhere. Dopo aver inviato i file dell'applicazione web che si trovano sulla directory locale del computer, dovreste avere una directory movieclassifier nel vostro account PythonAnywhere. Questa conterrà le stesse directory e gli stessi file della directory movieclassifier locale, come si può vedere nella Figura 9.8.

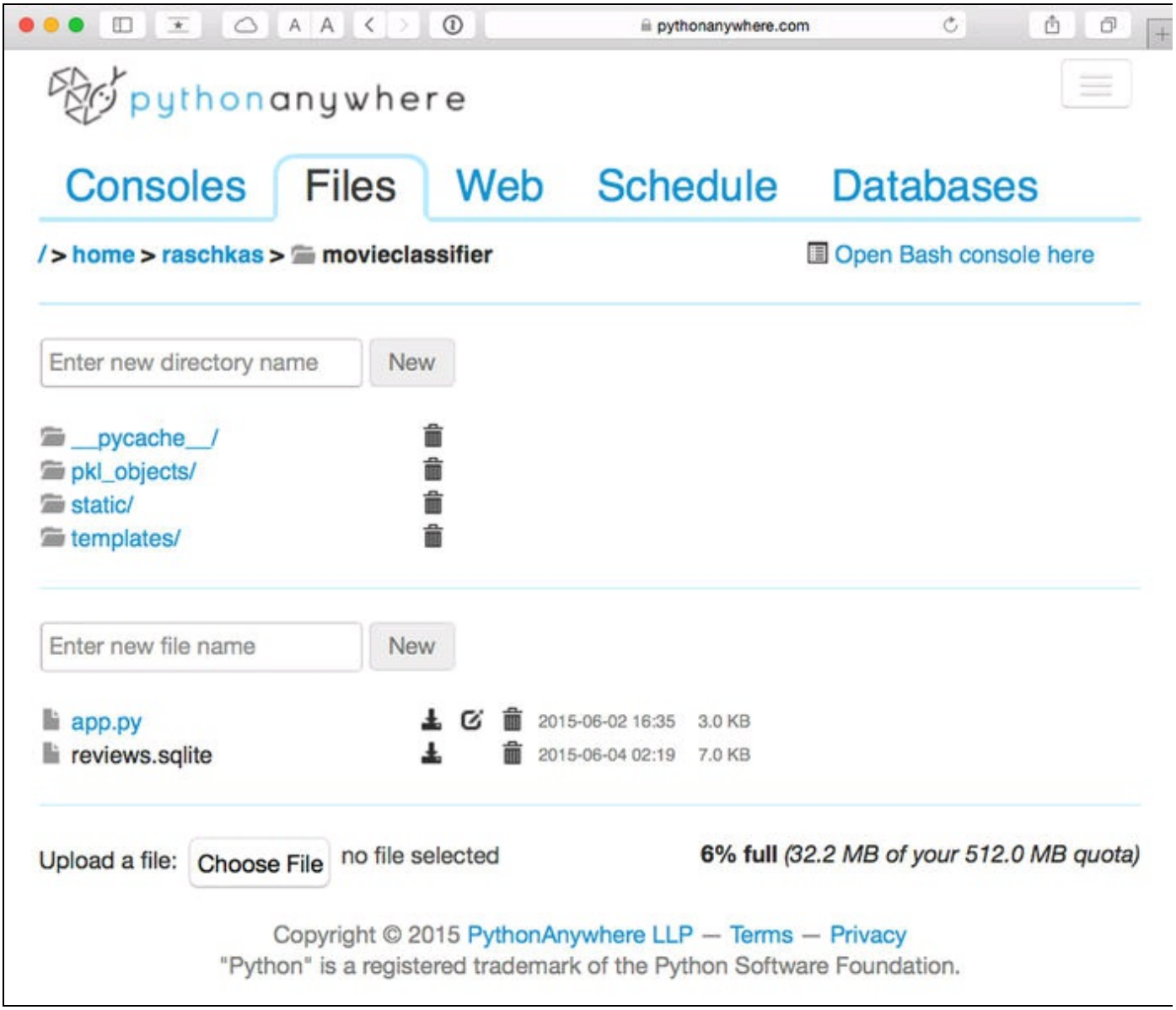


Figura 9.8

Infine richiamate ancora una volta la scheda *Web* e fate clic sul pulsante *Reload <nome-utente>.pythonanywhere.com*, per propagare le modifiche e aggiornare l'applicazione web. Alla fine l'applicazione web dovrebbe essere pronta all'uso e aperta al pubblico tramite l'indirizzo *<nome-utente>*.pythonanywhere.com.

NOTA

Sfortunatamente, i server web possono essere piuttosto sensibili ai piccoli problemi di un'applicazione web. Se state sperimentano problemi con l'esecuzione dell'applicazione web su PythonAnywhere e ricevete messaggi di errore nel browser, potete controllare i log degli errori del server, che possono essere consultati tramite la scheda *Web* dell'account PythonAnywhere, in modo da iniziare a individuare la causa del problema.

Aggiornamento del classificatore delle recensioni di film

Mentre il modello predittivo viene aggiornato al volo ogni volta che un utente fornisce un feedback sulla classificazione, gli aggiornamenti all'oggetto clf verranno reinizializzati ogni volta che il server web si blocca o viene riavviato. Se si richiama l'applicazione web, l'oggetto clf verrà reinizializzato dal file pickle classifier.pkl. Un'opzione per applicare gli aggiornamenti in modo permanente sarebbe quella di salvare (pickle) l'oggetto clf dopo ciascun aggiornamento. Tuttavia, questo sarebbe molto inefficiente dal punto di vista computazionale, con un numero crescente di utenti e potrebbe danneggiare il file qualora più utenti fornissero il proprio feedback simultaneamente. Una soluzione alternativa consiste nell'aggiornare il modello predittivo dai dati di feedback che vengono raccolti nel database SQLite. Un'opzione sarebbe quella di scaricare il database SQLite dal server PythonAnywhere, aggiornare localmente l'oggetto clf sul computer e poi inviare di nuovo il file a PythonAnywhere. Per aggiornare localmente il classificatore sul computer, creiamo un file script update.py nella directory movieclassifier con il seguente contenuto:

```
import pickle
import sqlite3
import numpy as np
import os

# import HashingVectorizer from local dir
from vectorizer import vect

def update_model(db_path, model, batch_size=10000):

    conn = sqlite3.connect(db_path)
    c = conn.cursor()
    c.execute('SELECT * from review_db')

    results = c.fetchmany(batch_size)
    while results:
        data = np.array(results)
        X = data[:, 0]
        y = data[:, 1].astype(int)
        classes = np.array([0, 1])
        X_train = vect.transform(X)
        model.partial_fit(X_train, y, classes=classes)
        results = c.fetchmany(batch_size)

    conn.close()
    return model

cur_dir = os.path.dirname(__file__)

clf = pickle.load(open(os.path.join(cur_dir,
                'pkl_objects',
                'classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

update_model(db_path=db, model=clf, batch_size=10000)

# Uncomment the following lines if you are sure that
# you want to update your classifier.pkl file
```

```
# permanently.

# pickle.dump(clf, open(os.path.join(cur_dir,
#             'pkl_objects', 'classifier.pkl'), 'wb')
#             , protocol=4)
```

La funzione update_model legge le voci del database SQLite in lotti di 10.000 voci per volta, a meno che il database contenga meno voci. Alternativamente, potreste leggere una voce alla volta utilizzando fetchone invece di fetchmany, il che però sarebbe molto inefficiente dal punto di vista computazionale. Utilizzando il metodo alternativo fetchall potrebbero sorgere problemi se si opera con grandi dataset che superano la capacità di memorizzazione del computer o del server.

Dopo aver creato lo script update.py, potete inviare anch'esso alla directory movieclassifier su PythonAnywhere e importare la funzione update_model nello script dell'applicazione principale app.py per aggiornare il classificatore dal database SQLite ogni volta che riavvierete l'applicazione web. Per farlo, dovete semplicemente aggiungere all'inizio del file app.py una riga di codice per importare la funzione update_model dallo script update.py:

```
# import update function from local dir
from update import update_model
```

Ora dovete richiamare la funzione update_model nel corpo dell'applicazione principale:

```
…
if __name__ == '__main__':
    clf = update_model(db_path="db", model=clf, batch_size=10000)
…
```

Riepilogo

In questo capitolo abbiamo trattato molti argomenti utili e pratici che estendono la conoscenza della teoria su cui si basano le attività di apprendimento automatico. Abbiamo imparato a serializzare un modello dopo l'addestramento e a caricarlo per ogni successivo utilizzo. Inoltre abbiamo creato un database SQLite per una memorizzazione efficiente dei dati e abbiamo creato un'applicazione web che ci consentisse di rendere disponibile al mondo esterno il nostro classificatore di film.

Nel corso di questo libro, abbiamo trattato molti concetti, molte tecniche e molti modelli con supervisione per la classificazione. Nel prossimo capitolo, esamineremo un'altra branca dell'apprendimento con supervisione, l'analisi a regressione, che ci consentirà di prevedere le variabili dei risultati su una scala continua, piuttosto che operare su etichette categoriche delle classi, come è accaduto nei modelli di classificazione sui quali abbiamo lavorato finora.

Previsioni di variabili target continue: l'analisi a regressione

Nel corso dei capitoli precedenti, abbiamo introdotto i concetti principali su cui si basa l'apprendimento con supervisione e abbiamo addestrato alcuni modelli per compiti di classificazione dove si trattava di prevedere l'appartenenza a gruppi o a categorie. In questo capitolo approfondiremo un'altra branca dell'apprendimento con supervisione: *l'analisi a regressione*.

I modelli a regressione vengono utilizzati per prevedere target variabili su scala *continua*, il che li rende interessanti per risolvere molte questioni in ambito scientifico e anche industriale, per esempio quando occorra trovare relazioni fra variabili, valutare tendenze o effettuare previsioni. Un esempio potrebbe essere la previsione delle vendite di un'azienda nei prossimi mesi.

In questo capitolo tratteremo i concetti principali dei modelli a regressione, affrontando i seguenti argomenti.

- Esplorazione e visualizzazione dei dataset.
- Approcci all'implementazione di modelli a regressione lineare.
- Addestramento dei modelli a regressione in grado di eliminare le anomalie.
- Valutazione dei modelli a regressione e diagnosi dei problemi più comuni.
- Adattamento dei modelli a regressione a dati non lineari.

Introduzione a un modello a regressione lineare semplice

L'obiettivo di un modello a regressione lineare semplice (*univariata*) consiste nel replicare le relazioni esistenti tra un'unica caratteristica (la variabile descrittiva *x*) e una *risposta* a valutazione continua (variabile target *y*). L'equazione di un modello lineare con una variabile descrittiva è definito nel seguente modo:

$$y = w_0 + w_1 x$$

Qui, il peso w_0 rappresenta il punto in cui viene intercettato l'asse *y* e w_1 è il coefficiente della variabile descrittiva. Il nostro obiettivo è quello di conoscere i pesi dell'equazione lineare per descrivere la relazione esistente fra la variabile descrittiva e la variabile target, che possiamo utilizzare per prevedere le risposte di nuove variabili descrittive, che non facevano parte del dataset di addestramento.

Sulla base dell'equazione lineare che abbiamo appena definito, la regressione lineare può essere considerata come la ricerca della migliore linea retta che attraversa i punti di campionamento, come possiamo vedere nella Figura 10.1.

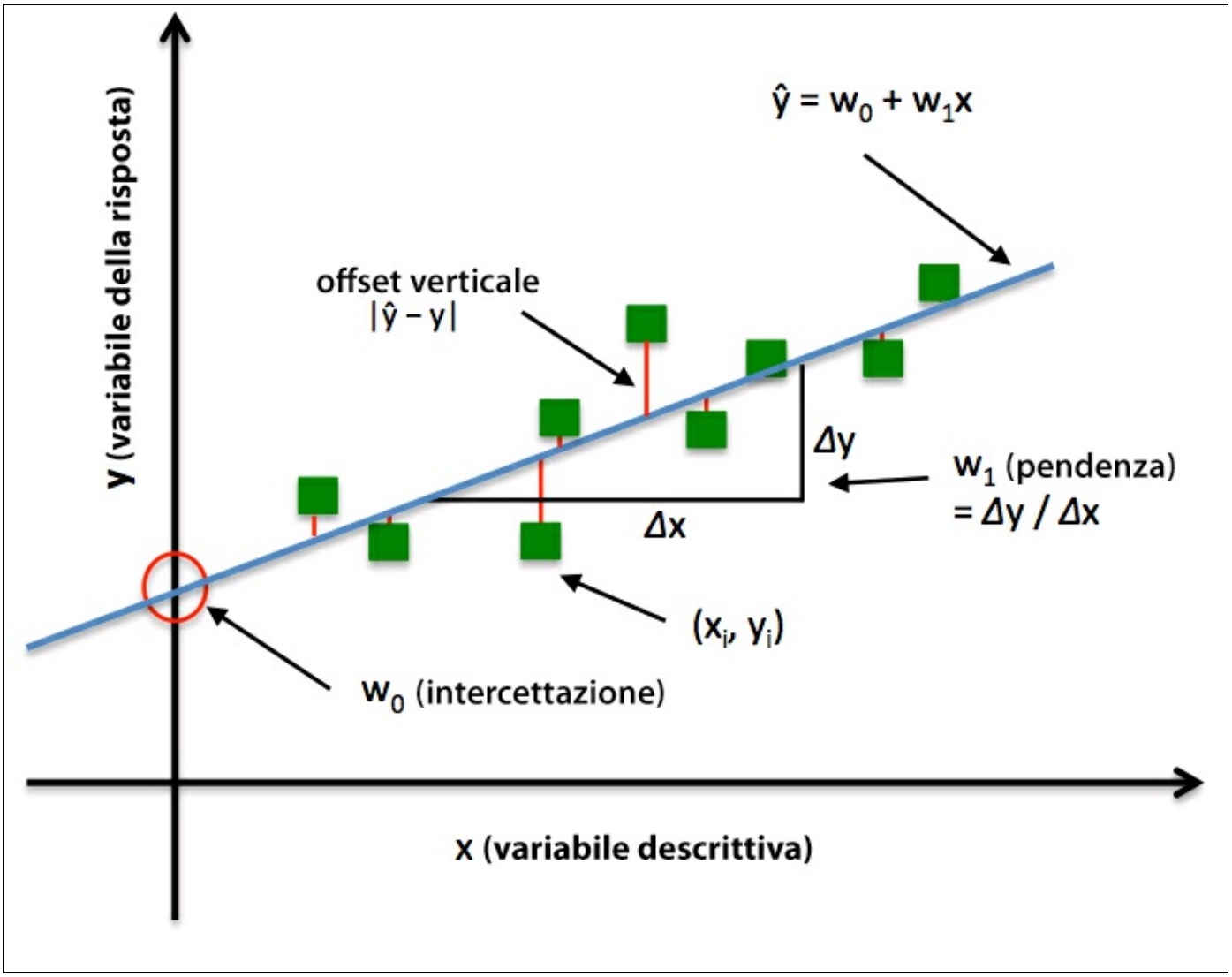


Figura 10.1

Questa linea è chiamata anche *linea di regressione* e le linee verticali che portano dalla linea di regressione ai punti campionati sono i cosiddetti *offset* o *residui*, gli errori della previsione. Il caso speciale di una sola variabile descrittiva è chiamato *espressione lineare semplice*, ma naturalmente possiamo anche generalizzare il modello a regressione lineare a più variabili descrittive.

Pertanto, questo processo è chiamato *regressione lineare multipla*:

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{i=0}^{m} w_i x_i = w^T x$$

Qui, w_0 è l'intercettazione dell'asse *y*, con $x_0 = 1$.

Esplorazione del dataset Housing

Prima di implementare il nostro primo modello a regressione lineare, introdurremo un nuovo dataset, *Housing Dataset*, che contiene informazioni sulle abitazioni dei dintorni di Boston, raccolte da D. Harrison e D.L. Rubinfeld nel 1978. Il dataset *Housing* è stato reso disponibile pubblicamente e può essere scaricato dal repository *UCI machine learning* all'indirizzo https://archive.ics.uci.edu/ml/datasets/Housing.

Le caratteristiche dei 506 campioni possono essere riassunte nel seguente modo.

- *CRIM:* tasso di criminalità *pro capite* per zona.
- *ZN:* proporzione di terreno residenziale per lotti maggiori di 25.000 piedi quadrati (circa 2300 m2).
- *INDUS:* proporzione di acri industriali non commerciali per città.
- *CHAS:* variabile fittizia Charles River (uguale a 1 se il tratto affianca il fiume, altrimenti è uguale a 0).
- *NOX:* concentrazione di ossido d'azoto (parti per 10 milioni).
- *RM:* numero medio di stanze per abitazione.
- *AGE:* proporzione delle unità abitate costruite prima del 1940.
- *DIS:* distanze pesate verso i cinque uffici di collocamento di Boston.
- *RAD:* indice di accessibilità rispetto alle grandi vie radiali di comunicazione.
- *TAX:* tasso di imposte sulla casa per 10.000 dollari.
- *PTRATIO:* rapporto allievi-docenti per città.
- *B:* calcolato come *1000(Bk - 0.63)^2*, dove *Bk* è la proporzione di persone di origine afroamericana.
- *LSTAT:* percentuale di popolazione con basso reddito.
- *MEDV:* valore mediano delle abitazioni di proprietà in migliaia di dollari.

Per il resto del capitolo, considereremo i prezzi delle abitazioni (MEDV) come la nostra variabile target, la variabile che vogliamo prevedere utilizzando una o più delle tredici variabili descrittive. Prima di esplorare ulteriormente questo dataset, leggiamolo dal repository UCI in un pandas DataFrame:

```
>>> import pandas as pd
>>> df = pd.read_csv(
...     'https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data',
...     header=None, sep='\s+')
>>> df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
...               'NOX', 'RM', 'AGE', 'DIS', 'RAD',
...               'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
>>> df.head()
```

Per confermare che il dataset sia stato caricato con successo, abbiamo visualizzato le prime cinque righe del dataset, riportate nella Figura 10.2.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

Figura 10.2

Visualizzazione delle caratteristiche importanti di un dataset

L'analisi esplorativa dei dati (*EDA – Exploratory Data Analysis*) è un primo passo importante, consigliato prima di addestrare un modello di machine learning. Nel resto di questo paragrafo, utilizzeremo alcune semplici ma utili tecniche che sono in dotazione nella "cassetta degli attrezzi" EDA e che possono aiutarci a rilevare visivamente la presenza di valori anomali, la distribuzione dei dati e le relazioni esistenti fra le caratteristiche le caratteristiche.

Innanzitutto, creiamo una *matrice a dispersione*, la quale ci consente di rappresentare le correlazioni a coppie fra le diverse caratteristiche di questo dataset. Per tracciare la matrice a dispersione, utilizzeremo la funzione pairplot della libreria seaborn (http://stanford.edu/~mwaskom/software/seaborn/), una libreria Python per il trattamento di grafici statistici, basata sul matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> sns.set(style='whitegrid', context='notebook')
>>> cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
>>> sns.pairplot(df[cols], size=2.5)
>>> plt.show()
```

Come possiamo vedere dalla Figura 10.3, la matrice a dispersione fornisce un utile riepilogo grafico delle relazioni esistenti nel dataset.

NOTA

L'importazione della libreria seaborn modifica l'estetica standard di matplotlib per la sessione corrente di Python. Se non volete utilizzare le impostazioni di stile di seaborn, potete eseguire il reset delle impostazioni di matplotlib tramite il seguente comando: >>> sns.reset_orig().

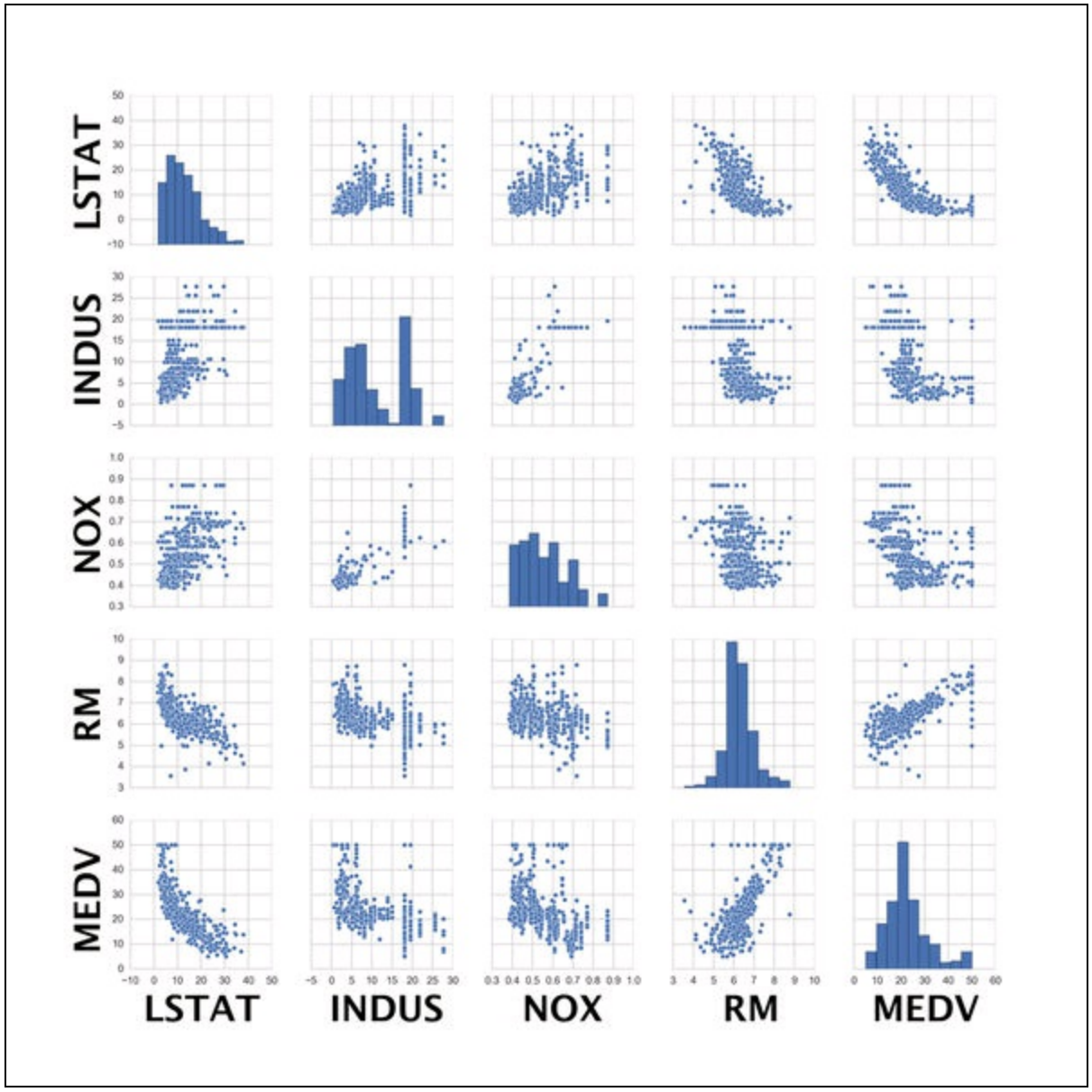


Figura 10.3

A causa dei limiti di spazio e per motivi di leggibilità, abbiamo tracciato solo cinque colonne del dataset: LSTAT, INDUS, NOX, RM e MEDV. Tuttavia, è consigliabile creare una matrice a dispersione dell'intero DataFrame, in modo da poter esplorare appieno i dati.

Utilizzando questa matrice a dispersione, possiamo rilevare rapidamente come sono distribuiti i dati e se contengono valori anomali. Per esempio, possiamo notare che esiste una relazione lineare fra RM e MEDV (quinta colonna, quarta riga). Inoltre, possiamo vedere nell'istogramma (il grafico in basso a destra nella matrice a dispersione) che la variabile MEDV sembra essere distribuita in modo normale, ma contiene numerosi valori anomali.

NOTA

Notate che, al contrario di quanto si possa ritenere, l'addestramento di un modello a regressione lineare non richiede che le variabili descrittiva o target abbiano una distribuzione normale. L'imposizione della normalità è un requisito solo per determinati test statistici e test di ipotesi che non rientrano però negli scopi di questo libro (D. C. Montgomery, E. A. Peck e G. G. Vining, *Introduction to linear regression analysis*, John Wiley and Sons 2012, pp. 318–319).

Per quantificare la relazione lineare fra le caratteristiche, creeremo ora una matrice di correlazione. Una matrice di correlazione è strettamente correlata alla matrice di covarianza che abbiamo visto nel paragrafo dedicato all'analisi del componente principale (*PCA – principal component analysis*) nel Capitolo 4,*Costruire buoni set di addestramento: la pre-elaborazione*. Intuitivamente, possiamo interpretare la matrice di correlazione come una versione in scala variata della matrice di covarianza. In realtà, la matrice di correlazione è identica a una matrice di covarianza, ma calcolata su dati standardizzati.

La matrice di correlazione è una matrice quadrata che contiene i *coefficienti di correlazione Pearson* (o *Pearson's r*) che misurano la dipendenza lineare fra coppie di caratteristiche. I coefficienti di correlazione sono limitati all'intervallo compreso fra -1 e 1. Due caratteristiche hanno una correlazione positiva perfetta se $r = 1$, non hanno alcuna correlazione se $r = 0$ e hanno una correlazione negativa perfetta se $r = -1$. Come abbiamo detto in precedenza, i coefficienti di correlazione di Pearson possono essere calcolati semplicemente come la covarianza fra le due caratteristiche x e y(a numeratore) divisa per il prodotto delle loro deviazioni standard (a denominatore):

$$r = \frac{\sum_{i=1}^{n}\left[\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)\right]}{\sqrt{\sum_{i=1}^{n}\left(x^{(i)} - \mu_x\right)^2}\sqrt{\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Qui, d denota la media del campione della caratteristica corrispondente, σ_{xy} è la covarianza fra le caratteristiche x e y e σ_x e σ_y sono le deviazioni standard delle due caratteristiche.

Approfondimento

Possiamo dimostrare che la covarianza fra caratteristiche standardizzate sia in realtà uguale al loro coefficiente di correlazione lineare.

Standardizzate innanzitutto le caratteristiche x e y, per ottenere i loro valori z, che possiamo denotare rispettivamente come x' e y':

$$x' = \frac{x - \mu_x}{\sigma_x}, y' = \frac{y - \mu_y}{\sigma_y}$$

Ricordate che possiamo calcolare la covarianza (della popolazione) fra due caratteristiche nel seguente modo:

$$\sigma_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Poiché la standardizzazione centra una variabile delle caratteristiche alla media 0, possiamo calcolare la covarianza fra le caratteristiche in scala nel seguente modo:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x' - 0\right)\left(y' - 0\right)$$

Tramite sostituzione, otteniamo il seguente risultato:

$$\frac{1}{n}\sum_{i}^{n}\left(\frac{x - \mu_x}{\sigma_x}\right)\left(\frac{y - \mu_y}{\sigma_y}\right)$$

$$\frac{1}{n \cdot \sigma_x \sigma_y}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Possiamo semplificare il tutto nel seguente modo:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Nel seguente esempio di codice, utilizzeremo la funzione corrcoef di NumPy sulle cinque colonne di caratteristiche che sono state precedentemente visualizzate nella matrice a dispersione e utilizzeremo la funzione heatmap di seaborn per tracciare l'array della matrice a correlazione come una mappa termica:

```
>>> import numpy as np
>>> cm = np.corrcoef(df[cols].values.T)
>>> sns.set(font_scale=1.5)
>>> hm = sns.heatmap(cm,
...         cbar=True,
...         annot=True,
...         square=True,
...         fmt='.2f',
...         annot_kws={'size': 15},
...         yticklabels=cols,
```

```
...             xticklabels=cols)
>>> plt.show()
```

Come possiamo vedere nella Figura 10.4, la matrice di correlazione ci fornisce un altro utile riepilogo grafico, che può aiutarci a selezionare le caratteristiche sulla base delle loro rispettive correlazioni lineari.

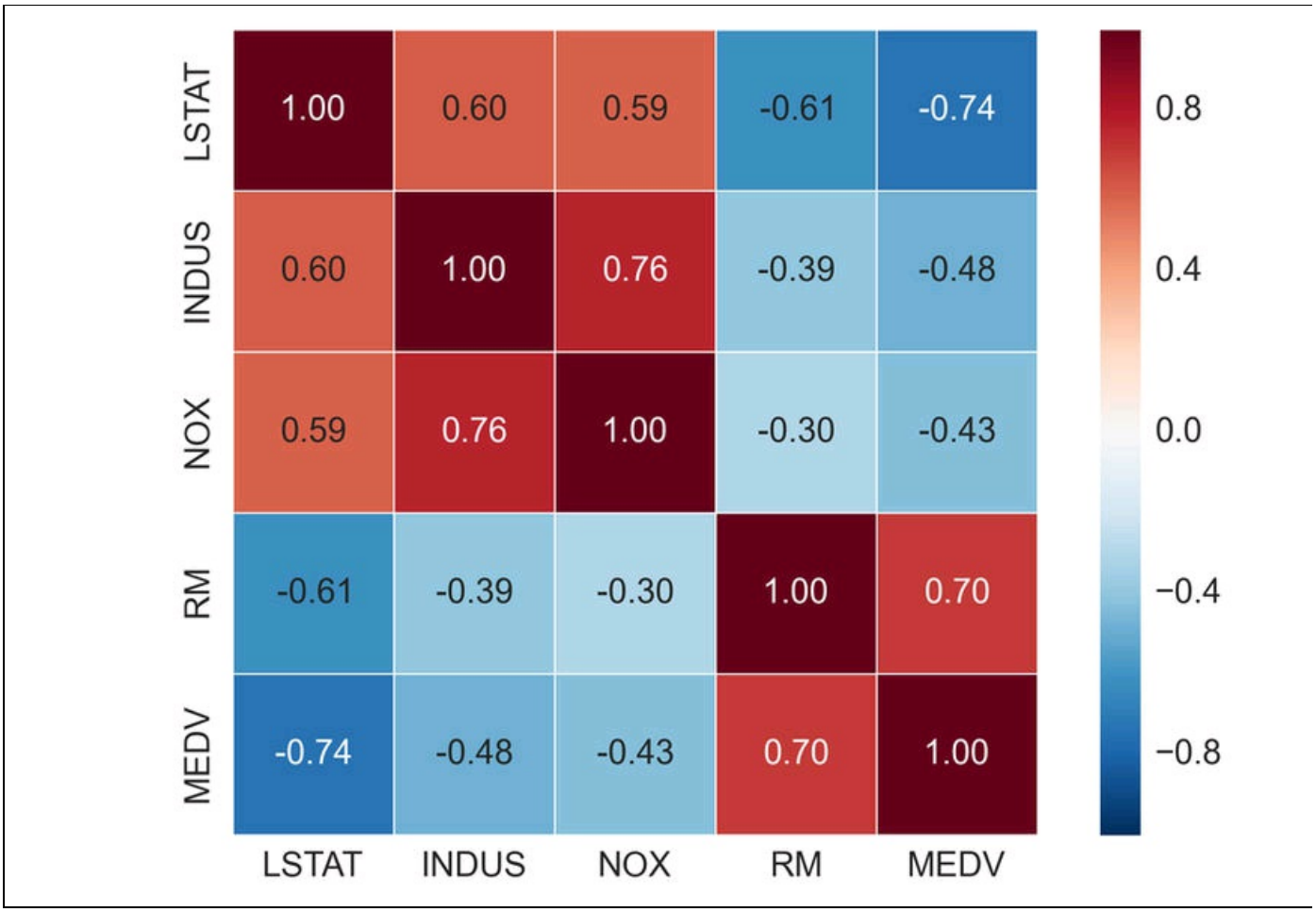


Figura 10.4

Per adattare un modello a regressione lineare, siamo interessati a quelle caratteristiche che hanno un'elevata correlazione con la variabile target MEDV. Osservando la matrice di correlazione precedente, vediamo che la variabile target MEDV mostra la correlazione più elevata con la variabile LSTAT (-0.74). Tuttavia, come ricorderete dalla matrice a dispersione, esiste una chiara relazione non lineare fra LSTAT e MEDV. D'altra parte, la correlazione fra RM e MEDV è anch'essa relativamente elevata (0.70) e data la relazione lineare esistente fra queste due variabili (che abbiamo osservato nel grafico a dispersione), RM sembra essere una buona scelta come variabile esplorativa per introdurre i concetti di un modello a regressione lineare semplice, cosa che faremo nel prossimo paragrafo.

Implementazione di un modello a regressione lineare OLS

All'inizio di questo capitolo, abbiamo detto che la regressione lineare può essere considerata come la ricerca della linea retta migliore che attraversa i punti campione dei dati di addestramento. Tuttavia, non abbiamo né definito il termine "migliore", né abbiamo trattato le varie tecniche di adattamento di tale modello. Nei prossimi paragrafi riempiremo le tessere mancanti di questo puzzle, utilizzando il metodo *OLS* (*Ordinary Least Squares*) per stimare i parametri della linea di regressione che minimizza la somma delle distanze verticali al quadrato (i residui o gli errori) nei punti di campionamento.

Risoluzione dei parametri di regressione nella discesa del gradiente

Riprendiamo la nostra implementazione dell'algoritmo *Adaline* (*ADAptive LInear NEuron*) trattata nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*. Ricordiamo che il neurone artificiale utilizza una funzione di attivazione lineare e che abbiamo definito una funzione di costo $J(\cdot)$, che abbiamo minimizzato per apprendere i pesi tramite degli algoritmi di ottimizzazione, come *Gradient Descent* (*GD*) e *Stochastic Gradient Descent* (*SGD*). Questa funzione di costo in Adaline si chiama *Sum of Squared Errors* (*SSE*). È identica alla funzione di costo OLS che abbiamo definito:

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Qui, $\hat{y}$ è il valore previsto $\hat{y} = w^T x$ (notate che il termine 1/2 viene utilizzato solo per comodità per derivare la regola di aggiornamento della GD). Sostanzialmente, la regressione lineare OLS può essere considerata come un Adaline senza la funzione del passo unitario, in modo che possiamo ottenere valori target continui, invece che etichette delle sole classi -1 e 1. Per illustrare questa analogia, prendiamo l'implementazione a discesa del gradiente (GD) di *Adaline* dal Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, ed eliminiamo la funzione del passo unitario per implementare il nostro primo modello a regressione lineare:

```
class LinearRegressionGD(object):
    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return self.net_input(X)
```

Per rinfrescare le idee sul modo in cui vengono aggiornati i pesi (fare un passo nella direzione opposta della pendenza) tornate a consultare il paragrafo dedicato ad Adaline nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*.

Per vedere in azione il nostro regressore `LinearRegressionGD`, utilizzeremo la variabile RM (numero di stanze) del dataset Housing quale variabile descrittiva per addestrare un modello che sia in grado di prevedere MEDV (il prezzo delle case). Inoltre, standardizzeremo le variabili per ottenere una migliore convergenza dell'algoritmo a discesa del gradiente. Il codice è il seguente:

```
>>> X = df[['RM']].values
>>> y = df['MEDV'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y)
>>> lr = LinearRegressionGD()
>>> lr.fit(X_std, y_std)
```

Nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, abbiamo detto che, quando utilizziamo gli algoritmi di ottimizzazione, come la discesa del gradiente, è sempre una buona idea tracciare il costo come una funzione del numero di epoch (passi di addestramento), con lo scopo di verificarne la convergenza. In sintesi, tracciamo il costo rispetto al numero di epoch, per controllare se la regressione lineare converge:

```
>>> plt.plot(range(1, lr.n_iter+1), lr.cost_)
>>> plt.ylabel('SSE')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

Come possiamo vedere nel grafico rappresentato nella Figura 10.5, l'algoritmo GD inizia a convergere dopo la quinta epoch.

Ora, visualizziamo la qualità con cui la linea della regressione lineare attraversa i dati di addestramento. Per farlo, definiremo una semplice funzione di supporto che

traccerà un grafico a dispersione dei campioni di addestramento e aggiungerà la linea di regressione:

```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='blue')
...     plt.plot(X, model.predict(X), color='red')
...     return None
```

Ora utilizzeremo la funzione `lin_regplot` per tracciare il numero di stanze rispetto al prezzo delle abitazioni:

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel('Average number of rooms [RM] (standardized)')
>>> plt.ylabel('Price in $1000\'s [MEDV] (standardized)')
>>> plt.show()
```

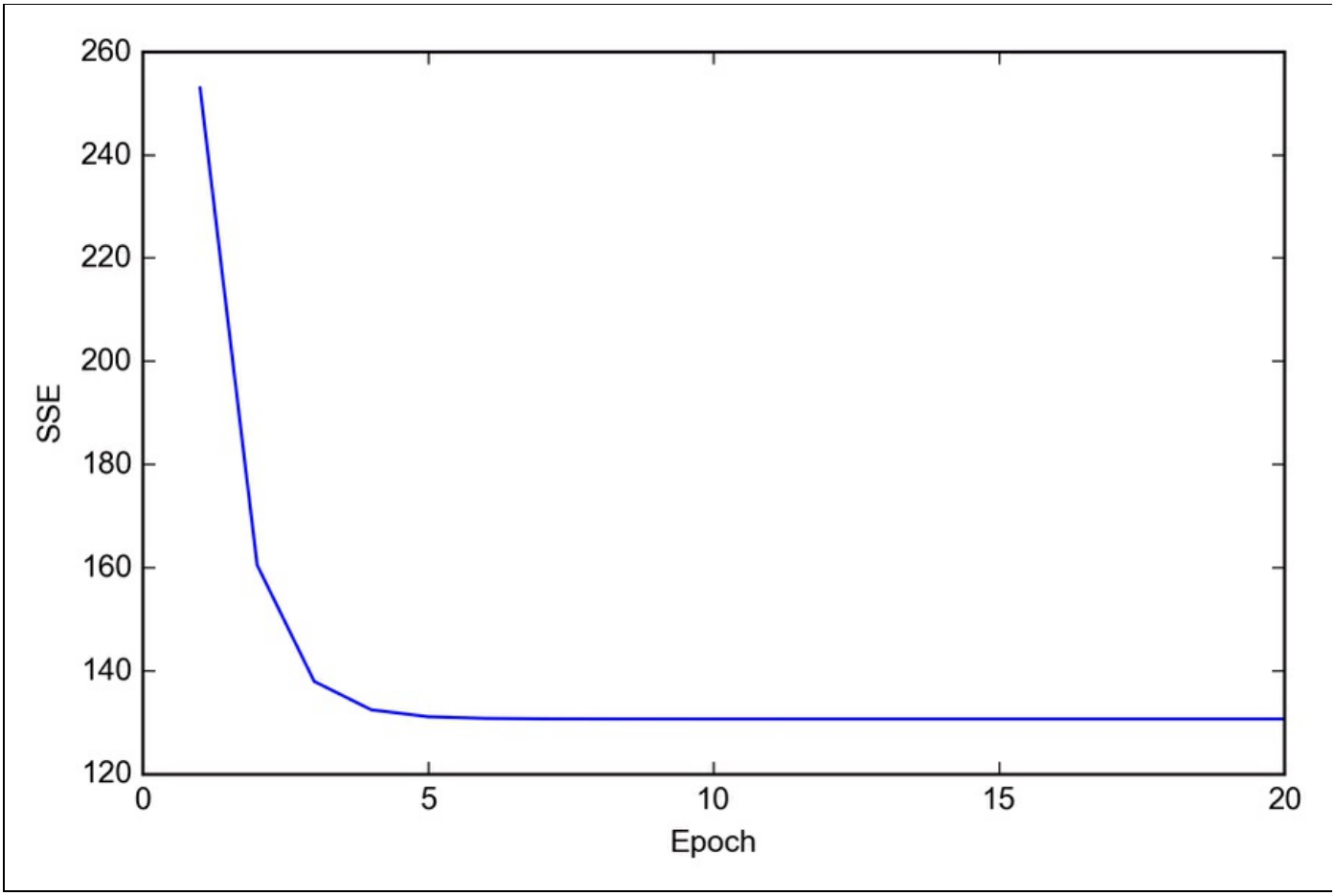


Figura 10.5

Come possiamo vedere nella Figura 10.6, la linea di regressione lineare riflette la tendenza generale, ovvero il fatto che il prezzo delle abitazioni tende ad aumentare con il numero delle stanze.

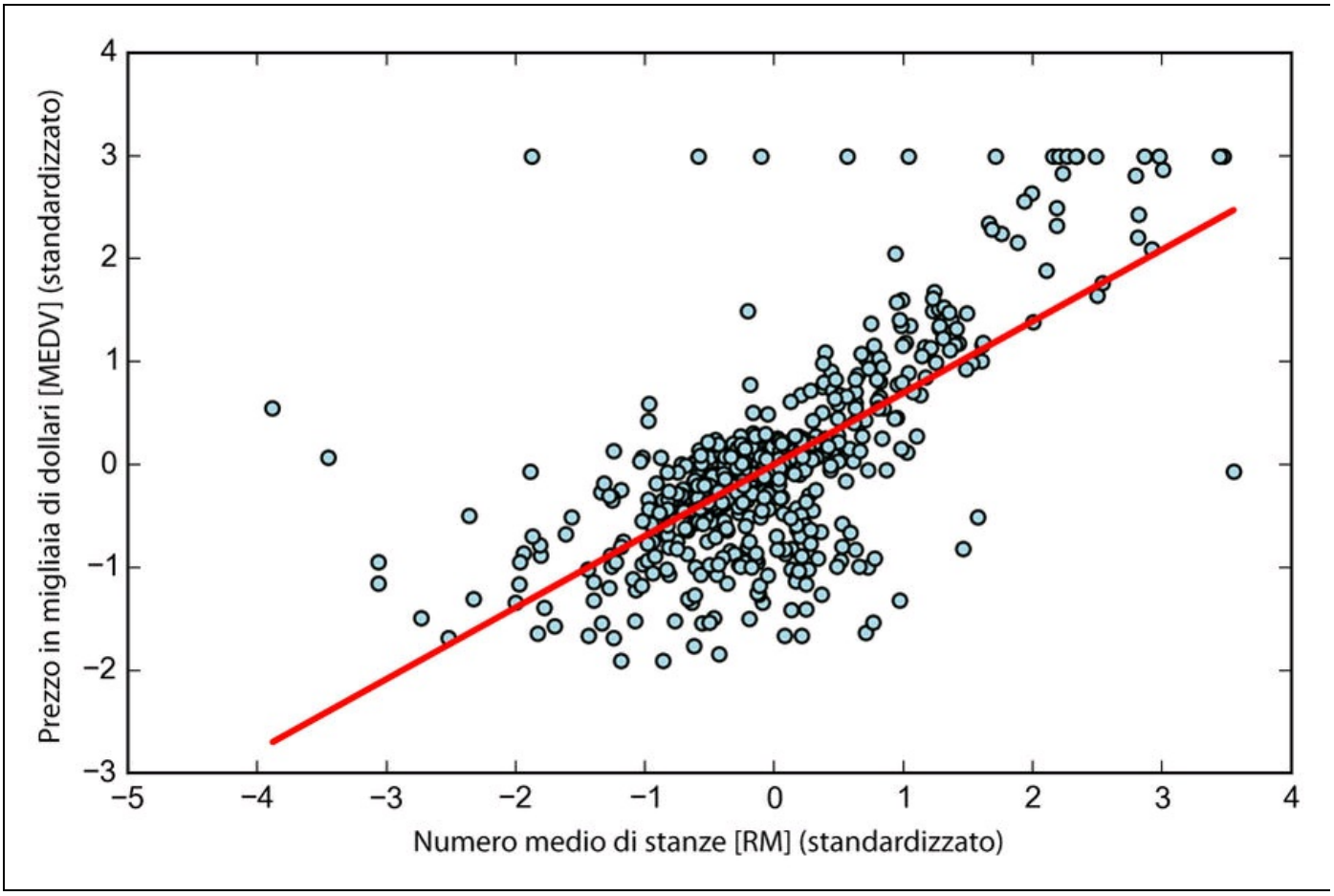


Figura 10.6

Sebbene questa osservazione abbia un senso anche dal punto di vista intuitivo, i dati ci dicono anche che il numero delle stanze non spiega appieno il prezzo delle abitazioni. Più avanti in questo capitolo vedremo come quantificare le prestazioni di un modello a regressione. È interessante notare che possiamo anche osservare una curiosa linea su $y = 3$ che suggerisce il fatto che i prezzi possono essere stati troncati. In alcune applicazioni, può anche essere importante rilevare le variabili previste nella loro scala originaria. Per calcolare la scala del prezzo risultante riportandolo sull'asse dei prezzi in migliaia di dollari, possiamo semplicemente aggiungere a `StandardScaler` il metodo `inverse_transform`:

```
>>> num_rooms_std = sc_x.transform([5.0])
>>> price_std = lr.predict(num_rooms_std)
>>> print("Price in $1000's: %.3f" % \
...       sc_y.inverse_transform(price_std))
Price in $1000's: 10.840
```

Nel precedente esempio di codice, abbiamo utilizzato il modello a regressione lineare precedentemente addestrato per prevedere il prezzo di un'abitazione di cinque stanze. Secondo il nostro modello, tale abitazione dovrebbe valere $10.840.

A margine, vale anche la pena di menzionare che, tecnicamente, non siamo costretti ad aggiornare i pesi dell'intercettazione se stiamo lavorando su variabili

standardizzate, in quanto in questi casi l'asse *y* viene intercettato sempre a 0. Possiamo confermarlo rapidamente stampando i pesi:

```
>>> print('Slope: %.3f' % lr.w_[1])
Slope: 0.695
>>> print('Intercept: %.3f' % lr.w_[0])
Intercept: -0.000
```

Stima del coefficiente di un modello regressione lineare tramite scikit-learn

Nel paragrafo precedente, abbiamo implementato un modello funzionante di analisi a regressione. Tuttavia, in un'applicazione reale, potremmo essere interessati a implementazioni più efficienti; per esempio, l'oggetto LinearRegression di scikit-learn che utilizza la libreria LIBLINEAR è un algoritmo di ottimizzazione avanzato che funzionano meglio con variabili non standardizzate. Questo può essere un vantaggio per alcune applicazioni:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> print('Slope: %.3f' % slr.coef_[0])
Slope: 9.102
>>> print('Intercept: %.3f' % slr.intercept_)
Intercept: -34.671
```

Come possiamo vedere eseguendo il codice precedente, il modello LinearRegression di scikit-learn adattato con le variabili RM e MEDV non standardizzate ha fornito coefficienti differenti. Confrontiamolo con la nostra implementazione a discesa del gradiente, tracciando MEDV rispetto a RM:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.show()w
```

Ora, quando tracciamo i dati di addestramento e il nostro modello adattato tramite il codice precedente, possiamo vedere (Figura 10.7) che il risultato generale sembra identico alla nostra implementazione a discesa del gradiente.

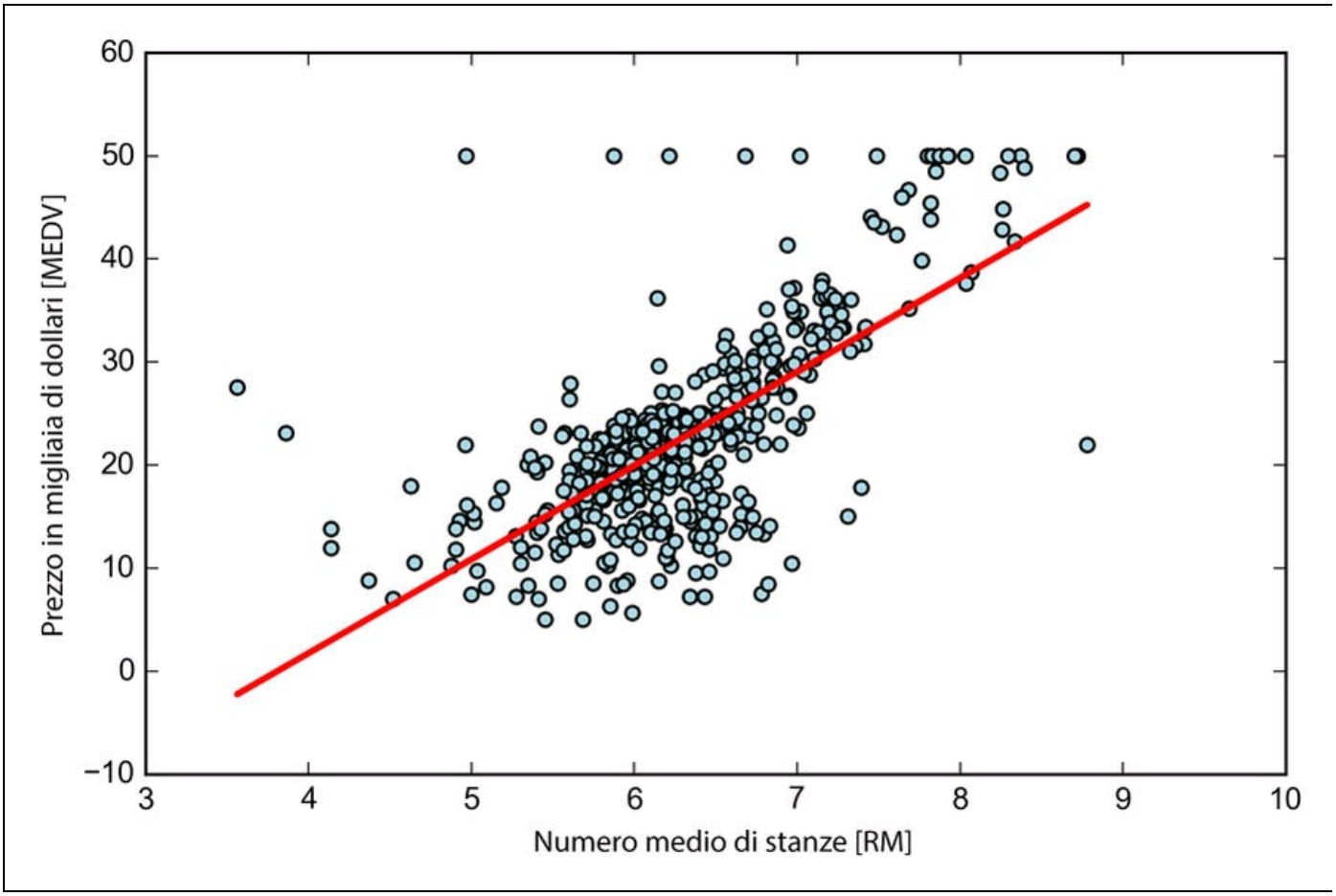


Figura 10.7

Approfondimento

Come alternativa all'uso delle librerie di machine learning, esiste anche una soluzione a formato chiuso per risolvere un problema OLS; tale soluzione coinvolge un sistema di equazioni lineari che può essere trovato nella maggior parte dei testi introduttivi all'analisi statistica:

$$w = \left(X^T X\right)^{-1} X^T y$$

Possiamo implementarla in Python nel seguente modo:

```
# adding a column vector of "ones"
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print('Slope: %.3f' % w[1])
Slope: 9.102
>>> print('Intercept: %.3f' % w[0])
Intercept: -34.671
```

Il vantaggio di questo metodo è che trova sicuramente la soluzione ottimale in modo analitico. Tuttavia, se stiamo utilizzando dataset di grandissime dimensioni, può essere semplicemente troppo costoso dal punto di vista computazionale invertire la matrice di questa (chiamata anche *equazione normale*) o la matrice campione può essere singolare (non invertibile), motivo per cui, in alcuni casi, può essere preferibile impiegare metodi iterativi. Se siete interessati al modo in cui ottenere le equazioni normali, si consiglia la lettura del capitolo del Dr. Stephen Pollock, *The*

Classical Linear Regression Model, dalle sue lezioni alla Università di Leicester, disponibili gratuitamente all'indirizzo http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf.

Adattamento di un solido modello a regressione utilizzando RANSAC

I modelli a regressione lineare possono avere un forte impatto dovuto alla presenza di valori anomali. In alcune situazioni, un piccolo sottoinsieme dei dati può avere un grosso effetto sui coefficienti stimati dal modello. Vi sono molti test statistici che possono essere impiegati per rilevare i valori anomali, ma questo argomento non rientra negli scopi di questo libro. Tuttavia, la rimozione dei valori anomali richiede sempre considerazioni approfondite in termini di esperienza nel campo dell'elaborazione dei dati e del dominio in questione.

Come alternativa per eliminare valori anomali, impiegheremo un metodo di regressione che utilizza l'algoritmo *RANSAC* (*RANdom SAmple Consensus*), che adatta un modello a regressione a un sottoinsieme dei dati, quelli che possiamo chiamare "in linea".

Possiamo riepilogare l'algoritmo RANSAC nel seguente modo.

1. Selezionare un numero casuale di campioni in linea e addestrare il modello.
2. Collaudare tutti gli altri punti dei dati rispetto al modello adattato e aggiungere quei punti che rientrano in una specifica tolleranza rispetto ai valori in linea.
3. Riadattare il modello utilizzando tutti i valori in linea.
4. Stimare l'errore del modello, adattato rispetto ai valori in linea.
5. Chiudere l'algoritmo se le prestazioni raggiungono una determinata soglia definita dall'utente o se è stato raggiunto un numero fisso di iterazioni, altrimenti tornare al Passo 1.

Dobbiamo dotare il modello lineare dell'algoritmo RANSAC utilizzando l'oggetto `RANSACRegressor` di scikit-learn:

```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(LinearRegression(),
...         max_trials=100,
...         min_samples=50,
...         residual_metric=lambda x: np.sum(np.abs(x), axis=1),
...         residual_threshold=5.0,
...         random_state=0)
>>> ransac.fit(X, y)
```

Impostiamo il numero massimo di iterazioni del `RANSACRegressor` a 100 e, utilizzando `min_samples=50`, impostiamo numero minimo di campioni scelti casualmente ad almeno 50. Utilizzando il parametro `residual_metric`, abbiamo fornito una funzione richiamabile `lambda` che calcola semplicemente le distanze verticali assolute fra la linea adattata e i

punti campione. Impostando il parametro residual_threshold a 5.0, consentiamo l'inclusione dei campioni nell'insieme dei valori in linea solo se la loro distanza verticale rispetto alla linea adattata rientra in 5 unità di distanza, un valore adatto a questo specifico dataset. Per default, scikit-learn utilizza la stima MAD (*Median Absolute Deviation*) per selezionare la soglia che considera i valori "in linea", fra i valori obiettivo y. Tuttavia, la scelta di un valore appropriato per la soglia dei valori in linea è specifica del problema, e questo è uno degli svantaggi di RANSAC. Sono stati sviluppati vari approcci nel corso degli ultimi anni per selezionare automaticamente una buona soglia per i valori in linea. Potete trovare una discussione dettagliata in R. Toldo e A. Fusiello, *Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting* (in "Image Analysis and Processing–ICIAP" 2009, pp. 123–131, Primavera 2009.

Dopo aver adattato il modello RANSAC, otteniamo i valori in linea e anomali dal modello a regressione lineare RANSAC adattato e li tracciamo insieme all'adattamento lineare:

```
>>> inlier_mask = ransac.inlier_mask_
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...             c='blue', marker='o', label='Inliers')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...             c='lightgreen', marker='s', label='Outliers')
>>> plt.plot(line_X, line_y_ransac, color='red')
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Come possiamo vedere nella Figura 10.8, il modello a regressione lineare è stato adattato sul set rilevato dei valori in linea, rappresentati dai cerchi.

Quando stampiamo la pendenza e il punto di intercettazione del modello tramite il codice seguente, possiamo vedere che la linea della regressione lineare è leggermente differente dall'adattamento che abbiamo ottenuto nel paragrafo precedente senza RANSAC:

```
>>> print('Slope: %.3f' % ransac.estimator_.coef_[0])
Slope: 9.621
>>> print('Intercept: %.3f' % ransac.estimator_.intercept_)
Intercept: -37.137
```

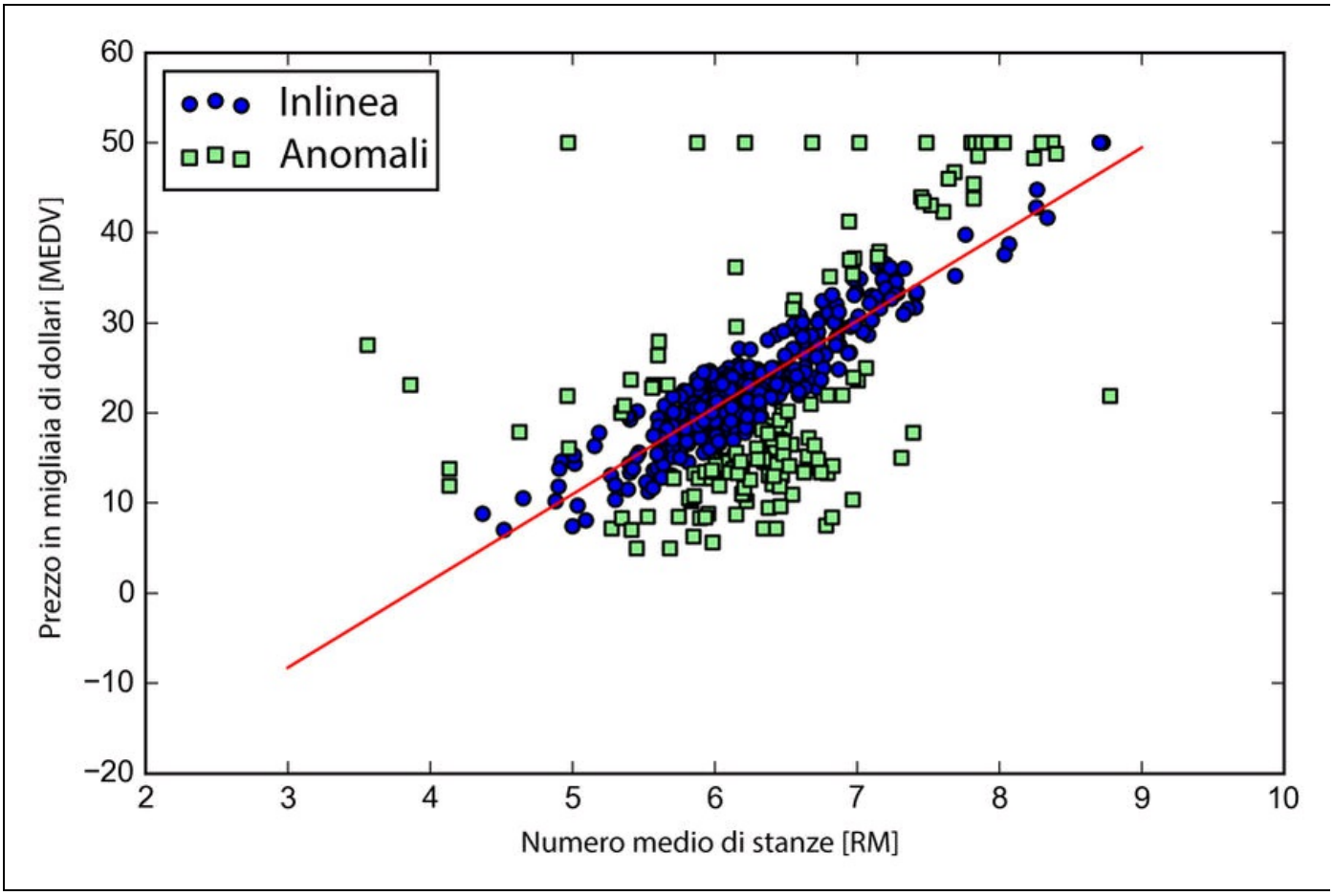


Figura 10.8

Utilizzando RANSAC abbiamo ridotto l'effetto potenziale dei valori anomali su questo dataset, ma non sappiamo se questo approccio ha avuto un effetto positivo sulle prestazioni delle previsioni su dati imprevisti. Pertanto, nel prossimo paragrafo parleremo della valutazione di un modello a regressione per approcci differenti, un elemento fondamentale della costruzione di sistemi per la modellazione predittiva.

Valutazione delle prestazioni dei modelli a regressione lineare

Nel paragrafo precedente, abbiamo visto come adattare un modello a regressione su dati di addestramento. Tuttavia, nei capitoli precedenti abbiamo imparato che è fondamentale collaudare il modello su dati che non siano ancora stati esaminati durante la fase di addestramento, in modo da ottenere una stima non viziata delle sue prestazioni.

Come ricordiamo dal Capitolo 6, *Valutazione dei modelli e ottimizzazione degli iperparametri*, vogliamo suddividere il nostro dataset in dataset di addestramento e di test, dove utilizziamo il primo per adattare il modello e il secondo per valutare le sue prestazioni di generalizzazione su dati mai visti prima. Invece di procedere con il modello a regressione semplice, utilizzeremo ora tutte le variabili del dataset e addestreremo un modello a regressione multipla:

```
>>> from sklearn.cross_validation import train_test_split
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...      X, y, test_size=0.3, random_state=0)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

Poiché il nostro modello utilizza più variabili descrittive, non possiamo rappresentare la linea di regressione lineare (per essere precisi sarebbe un iperpiano) in un grafico bidimensionale, ma possiamo stampare i residui (le differenze o le distanze verticali fra il valore effettivo e il valore previsto) rispetto ai valori previsti per diagnosticare il nostro modello a regressione. Questi grafici residui sono un metodo di analisi grafica comunemente utilizzato per diagnosticare il funzionamento dei modelli a regressione, in modo da individuare problemi di linearità e valori anomali e per controllare se gli errori sono distribuiti in modo casuale.

Utilizzando il codice seguente, tracciamo un grafico dei residui, dove semplicemente sottraiamo le vere variabili target dai responsi previsti:

```
>>> plt.scatter(y_train_pred, y_train_pred - y_train,
...            c='blue', marker='o', label='Training data')
>>> plt.scatter(y_test_pred,  y_test_pred - y_test,
...            c='lightgreen', marker='s', label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

Dopo aver eseguito il codice, dovremmo vedere un grafico dei residui con una linea che passa attraverso l'origine dell'asse *x* (Figura 10.9)

Nel caso di una previsione perfetta, i residui sarebbero esattamente a 0, una situazione che probabilmente non incontreremo mai in un'applicazione pratica. Tuttavia, per un buon modello a regressione, ci aspetteremmo che gli errori siano distribuiti casualmente e che i residui siano dispersi altrettanto casualmente attorno alla linea centrale. Se vediamo comparire degli schemi in un grafico dei residui, significa che il modello non è stato in grado di catturare alcune informazioni descrittive, che sono sfuggite nei residui, come possiamo vedere nel grafico rappresentato nella Figura 10.9. Inoltre possiamo utilizzare i grafici dei residui per rilevare i valori anomali, che sono rappresentati dai punti che presentano una notevole deviazione rispetto alla linea centrale.

Un'altra misura quantitativa utile delle prestazioni del modello è il cosiddetto *MSE* (*Mean Squared Error*), che è semplicemente il valore medio della funzione di costo SSE minimizzate per adattarla al modello a regressione lineare. L'errore MSE è utile per confrontare diversi modelli a regressione, con lo scopo di ottimizzare i loro parametri tramite una ricerca a griglia e una convalida incrociata:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

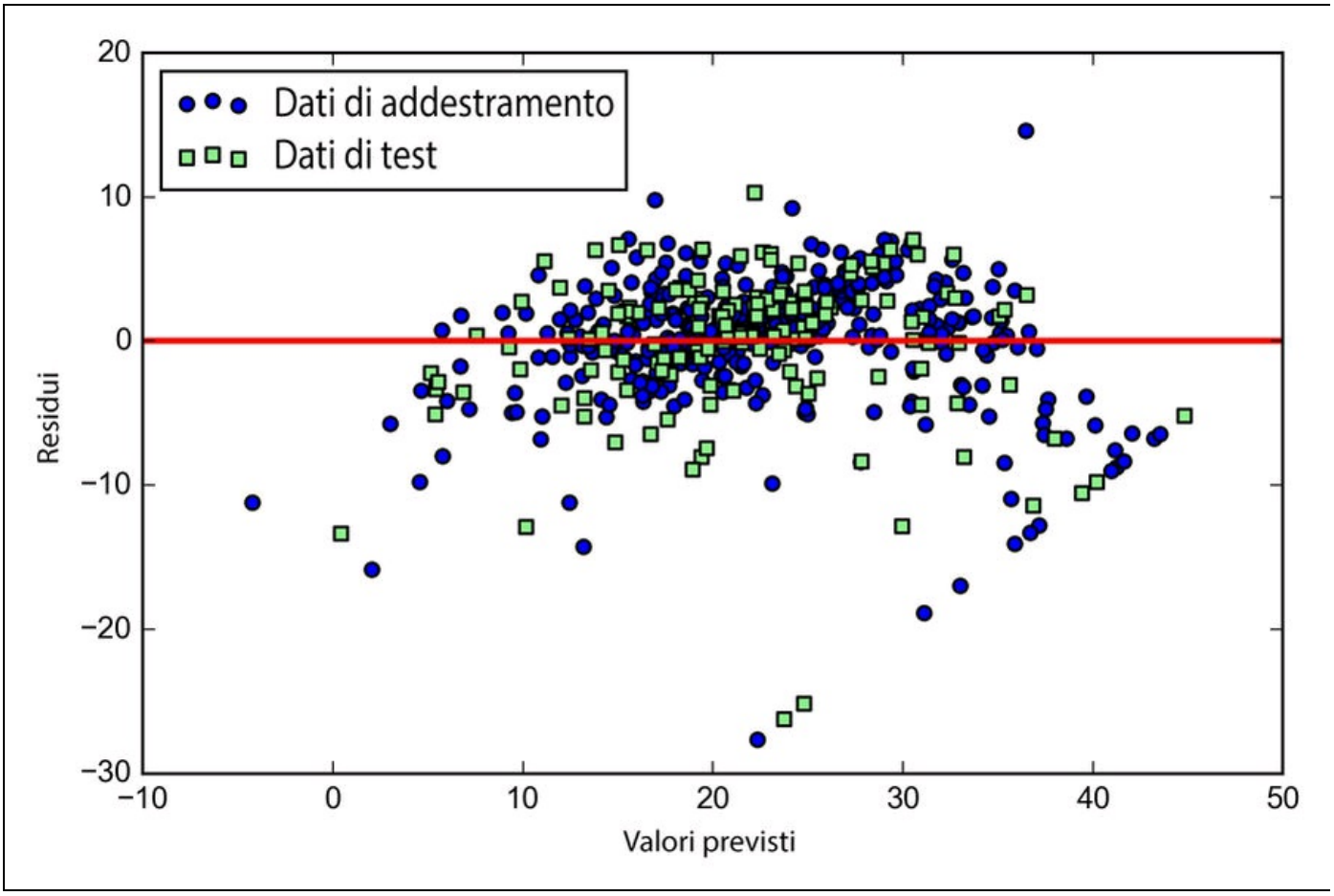


Figura 10.9

Eseguite il codice seguente:

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
```

Vedremo che l'errore MSE sul set di addestramento è pari a 19.96 e l'errore MSE sul set di test è molto più ampio, 27.20, il che è un indicatore che il nostro modello ha problemi di overfitting sui dati di addestramento.

Talvolta può essere più utile rilevare il coefficiente di determinazione (R^2), che può essere considerato una versione standardizzata dell'errore MSE, e che fornisce una migliore interpretabilità delle prestazioni del modello. In altre parole, R^2è la frazione della varianza nella risposta catturata dal modello. Il valore R^2 è definito nel seguente modo:

$$R^2 = 1 - \frac{SSE}{SST}$$

Qui, SSE è la somma degli errori al quadrato e SST è la somma totale dei quadrati $SST = \sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2$ o, in altre parole, è semplicemente la varianza della risposta.

Mostriamo rapidamente che R^2, in realtà, è semplicemente una versione in scala diversa dell'errore MSE:

$$R^2 = 1 - \frac{SSE}{SST}$$

$$1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}$$

$$1 - \frac{MSE}{Var(y)}$$

Per il dataset di addestramento, R^2 è compreso fra 0 e 1, ma può diventare negativo per il set di test. Se $R^2 = 1$, il modello individua perfettamente i dati e il corrispondente MSE è uguale a 0.

Valutato sui dati di addestramento, il valore R^2 del nostro modello è pari a 0.765, che non sembra poi così male. Tuttavia, R^2 sul dataset di test è pari solo a 0.673, come possiamo rilevare tramite il codice seguente:

```
>>> from sklearn.metrics import r2_score
>>> print('R^2 train: %.3f, test: %.3f' %
...       (r2_score(y_train, y_train_pred),
...        r2_score(y_test, y_test_pred)))
```

Uso di metodi regolarizzati per la regressione

Come abbiamo descritto nel Capitolo 3, *I classificatori di machine learning di scikit-learn*, la regolarizzazione è un approccio che risolve il problema dell'overfitting aggiungendo ulteriori informazioni: si tratta di ridurre i valori dei parametri del modello, in modo da introdurre una penalità alla complessità. Gli approcci più utilizzati per ottenere una regressione lineare regolarizzata sono i metodi *Ridge Regression, LASSO* (*Least Absolute Shrinkage and Selection Operator*) ed *Elastic Net*.

Ridge Regression è un modello con penalizzazione L2, dove aggiungiamo la somma al quadrato dei pesi alla nostra funzione di costo:

$$J(w)_{Ridge} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \| w \|_2^2$$

Qui:

$$L2: \quad \lambda \| w \|_2^2 = \lambda \sum_{j=1}^{m} w_j^{\,2}$$

Incrementando il valore dell'iperparametro λ, aumentiamo l'intensità di regolarizzazione e riduciamo i pesi del nostro modello. Notate che non regolarizziamo il termine di intercettazione, w_0.

Un approccio alternativo, che può condurre a modelli sparsi è *LASSO*. A seconda dell'intensità della regolarizzazione, alcuni valori possono divenire uguali a 0, il che rende LASSO utile anche come tecnica di selezione delle caratteristiche con supervisione:

$$J(w)_{LASSO} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \| w \|_1$$

Qui:

$$L1 : \lambda \| w \|_1 = \lambda \sum_{j=1}^{m} \left| w_j \right|$$

Un limite di LASSO è il fatto che seleziona al massimo n variabili, se m > n. Un compromesso fra la Ridge Regression e LASSO è *Elastic Net*, che introduce una penalità L1 per generare la sparsità e una penalità L2 per superare alcuni dei limiti di LASSO, come il numero delle variabili selezionate.

$$J(w)_{ElasticNet} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda_1 \sum_{j=1}^{m} w_j^{\,2} + \lambda_2 \sum_{j=1}^{m} \left| w_j \right|$$

Questi modelli a regressione regolarizzata sono tutti disponibili in scikit-learn e il loro uso è simile al modello a regressione standard, tranne per il fatto che dobbiamo specificare l'intensità della regolarizzazione tramite il parametro λ, per esempio, ottimizzato tramite una convalida incrociata a k parti.

Un modello Ridge Regression può essere inizializzato nel seguente modo:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Notate che l'intensità della regolarizzazione è regolata da `alpha`, che è simile al parametro λ. Analogamente, potremmo inizializzare un regressore LASSO dal sottomodello `linear_model`:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Infine, l'implementazione `ElasticNet` ci consente di variare il rapporto fra L1 e L2:

```
>>> from sklearn.linear_model import ElasticNet
>>> lasso = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

Se impostiamo `l1_ratio` a `1.0`, il regressore `ElasticNet` diventa uguale al regressore LASSO. Per informazioni più dettagliate sulle varie implementazioni della regressione lineare, potete consultare la documentazione di `linear_model` all'indirizzo http://scikit-learn.org/stable/modules/linear_model.html.

Trasformare un modello a regressione lineare in uno a regressione a curva polinomiale

Nei paragrafi precedenti, abbiamo presupposto che esistesse una relazione lineare fra le variabili descrittive e di risposta. Un modo per considerare la violazione di questo assunto di linearità consiste nell'utilizzare un modello a regressione polinomiale, aggiungendo dei termini polinomiali:

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d$$

Qui, d denota il grado del polinomio. Sebbene possiamo utilizzare la regressione polinomiale per modellare una relazione non lineare, questo viene comunque considerato un modello a regressione lineare multipla, a causa dei coefficienti di regressione lineare w.

Ora vedremo come utilizzare la classe trasformata PolynomialFeatures di scikit-learn per aggiungere un termine quadratico ($d = 2$) a un problema di regressione semplice con una variabile descrittiva e poi confronteremo l'adattamento polinomiale con quello lineare. I passi da utilizzare sono i seguenti.

1. Aggiungere un termine polinomiale di secondo grado:

```
from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([258.0, 270.0, 294.0,
…                 320.0, 342.0, 368.0,
…                 396.0, 446.0, 480.0,
…                 586.0])[:, np.newaxis]
>>> y = np.array([236.4, 234.4, 252.8,
…                 298.6, 314.2, 342.2,
…                 360.8, 368.0, 391.2,
…                 390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

2. Adattare un modello regressione lineare semplice, per il confronto:

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250,600,10)[:, np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

3. Adattare un modello a regressione multipla sulle caratteristiche trasformate per la regressione polinomiale:

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
Plot the results:
>>> plt.scatter(X, y, label='training points')
>>> plt.plot(X_fit, y_lin_fit,
```

```
...         label='linear fit', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...         label='quadratic fit')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Nel grafico rappresentato nella Figura 10.10, possiamo vedere che l'adattamento polinomiale cattura la relazione tra la variabile di risposta e quella descrittiva molto meglio rispetto all'adattamento lineare.

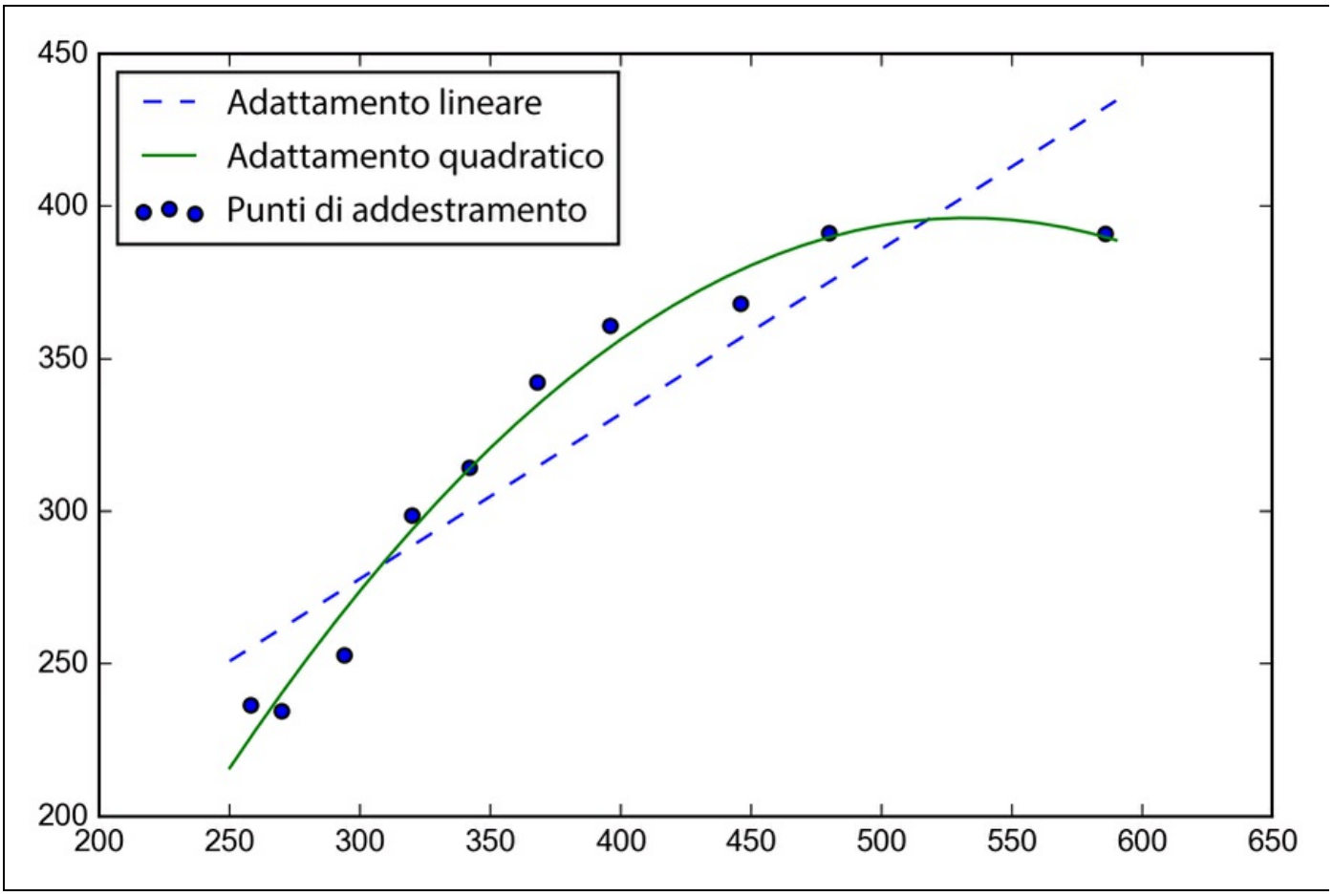


Figura 10.10

```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> print('Training MSE linear: %.3f, quadratic: %.3f' % (
...         mean_squared_error(y, y_lin_pred),
...         mean_squared_error(y, y_quad_pred)))
Training MSE linear: 569.780, quadratic: 61.330
>>> print('Training R^2 linear: %.3f, quadratic: %.3f' % (
...         r2_score(y, y_lin_pred),
...         r2_score(y, y_quad_pred)))
Training R^2 linear: 0.832, quadratic: 0.982
```

Come possiamo vedere dopo aver eseguito il codice precedente, l'errore MSE è diminuito da 570 (adattamento lineare) a 61 (adattamento quadratico) e il coefficiente di determinazione riflette il migliore adattamento del modello quadratico ($R^2 = 0.982$) rispetto all'adattamento lineare ($R^2 = 0.832$) in questo specifico problema "giocattolo".

Modellazione di relazioni non lineari nel dataset Housing

Dopo aver visto come costruire caratteristiche polinomiali per adattare relazioni non lineari in un semplice problema-giocattolo, vediamo un esempio più concreto e applichiamo questi concetti ai dati contenuti nel dataset *Housing*. Tramite il codice seguente, modelleremo la relazione esistente fra il prezzo delle abitazioni e il valore LSTAT (percent lower status of the population) utilizzando polinomi di secondo grado (quadratici) e di terzo grado (cubici), che confronteremo con un adattamento lineare.

Il codice è il seguente:

```
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> regr = LinearRegression()
# create polynomial features
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)
# linear fit
>>> X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]
>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))
# quadratic fit
>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))
# cubic fit
>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))
# plot results
>>> plt.scatter(X, y,
...             label='training points',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label='linear (d=1), $R^2=%.2f$'
...            % linear_r2,
...          color='blue',
...          lw=2,
...          linestyle=':')
>>> plt.plot(X_fit, y_quad_fit,
...          label='quadratic (d=2), $R^2=%.2f$'
...            % quadratic_r2,
...          color='red',
...          lw=2,
...          linestyle='-')
>>> plt.plot(X_fit, y_cubic_fit,
...          label='cubic (d=3), $R^2=%.2f$'
...            % cubic_r2,
...          color='green',
...          lw=2,
...          linestyle='--')
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.legend(loc='upper right')
>>> plt.show()
```

Come si può vedere nella Figura 10.11, l'adattamento cubico cattura la relazione fra i prezzi delle abitazioni e LSTAT meglio rispetto all'adattamento lineare quadratico. Tuttavia, dobbiamo anche considerare che l'incremento del grado

polinomiale aumenta la complessità del modello e pertanto anche le probabilità che si sviluppi un overfitting. Pertanto, nella pratica, è sempre consigliabile valutare le prestazioni del modello su dataset di test distinti, in modo da stimare le loro prestazioni di generalizzazione.

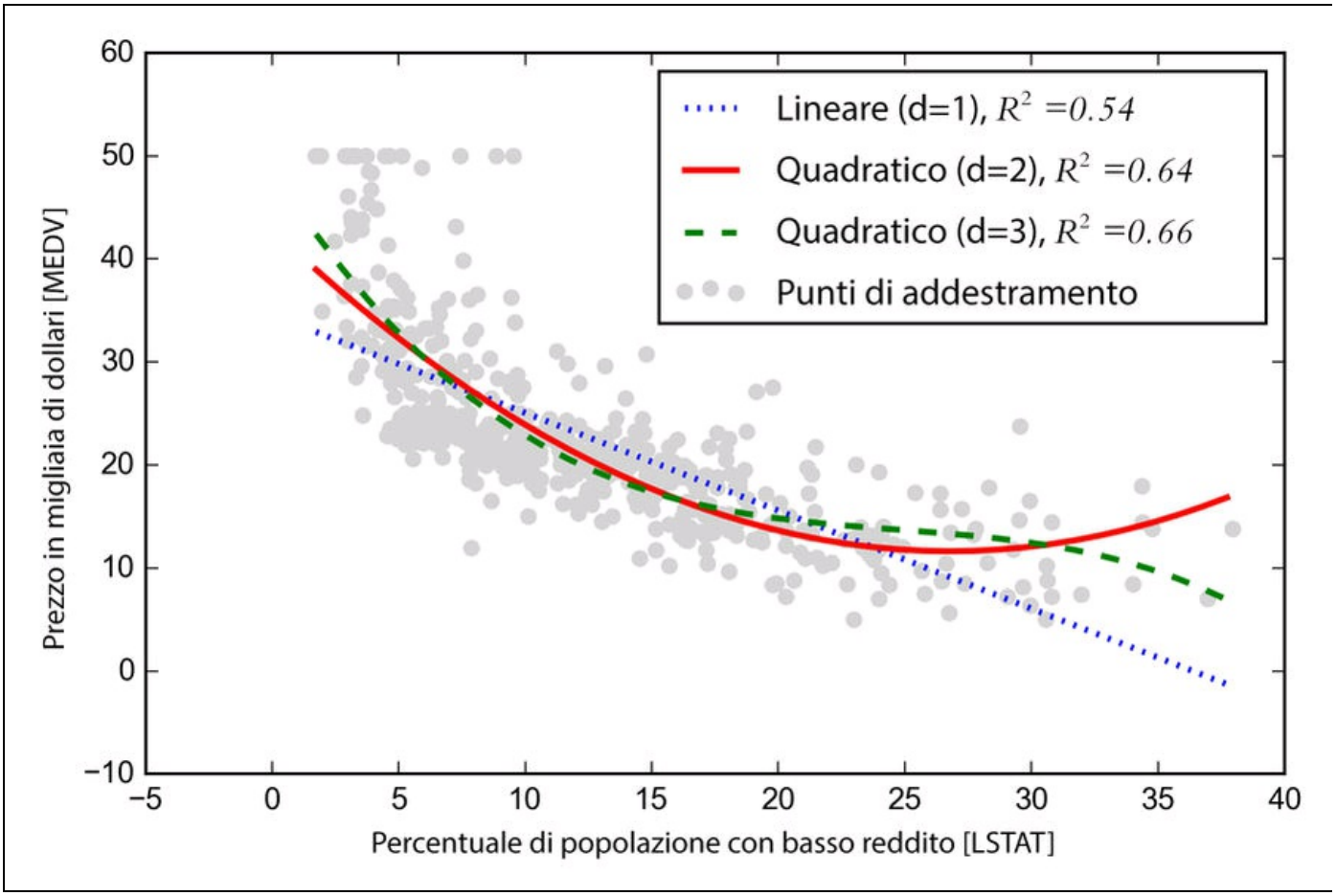


Figura 10.11

Inoltre, gli elementi polinomiali non sono sempre la scelta migliore per modellare le relazioni non lineari. Per esempio, semplicemente osservando il grafico a dispersione MEDV-LSTAT, potremmo immaginare che una trasformazione logaritmica della variabile della caratteristica LSTAT e la radice quadrata di MEDV possano proiettare i dati su uno spazio di caratteristiche lineare, che si presta quindi a un adattamento a regressione lineare. Verifichiamo questa ipotesi tramite il codice seguente:

```
# transform features
>>> X_log = np.log(X)
>>> y_sqrt = np.sqrt(y)
# fit features
>>> X_fit = np.arange(X_log.min()-1,
...                   X_log.max()+1, 1)[:, np.newaxis]
>>> regr = regr.fit(X_log, y_sqrt)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y_sqrt, regr.predict(X_log))
# plot results
>>> plt.scatter(X_log, y_sqrt,
...             label='training points',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
```

```
...         label='linear (d=1), $R^2=%.2f$' % linear_r2,
...         color='blue',
...         lw=2)
>>> plt.xlabel('log(% lower status of the population [LSTAT])')
>>> plt.ylabel('$\sqrt{Price \; in \; \$1000\'s [MEDV]}$')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Dopo aver trasformato la variabile descrittiva nello spazio logaritmico e aver calcolato la radice quadrata delle variabili target, siamo in grado di catturare la relazione esistente fra le due variabili con una regressione lineare, che sembra adattarsi particolarmente bene ai dati ($R^2 = 0.69$), meglio di tutte le trasformazioni polinomiali delle caratteristiche esaminate in precedenza (Figura 10.12).

Risoluzione delle relazioni non lineari tramite foreste casuali

In questo paragrafo esamineremo la regressione a *foresta casuale*, che è concettualmente differente rispetto agli altri modelli a regressione di questo capitolo. Una foresta casuale, che è un insieme di più *alberi decisionali*, può essere considerata come la somma di più funzioni lineari, rispetto invece ai modelli a regressione lineare e polinomiale globali che abbiamo appena visto. In altre parole, tramite l'algoritmo ad albero decisionale, suddividiamo lo spazio di input in regioni più compatte, che divengono pertanto anche più gestibili.

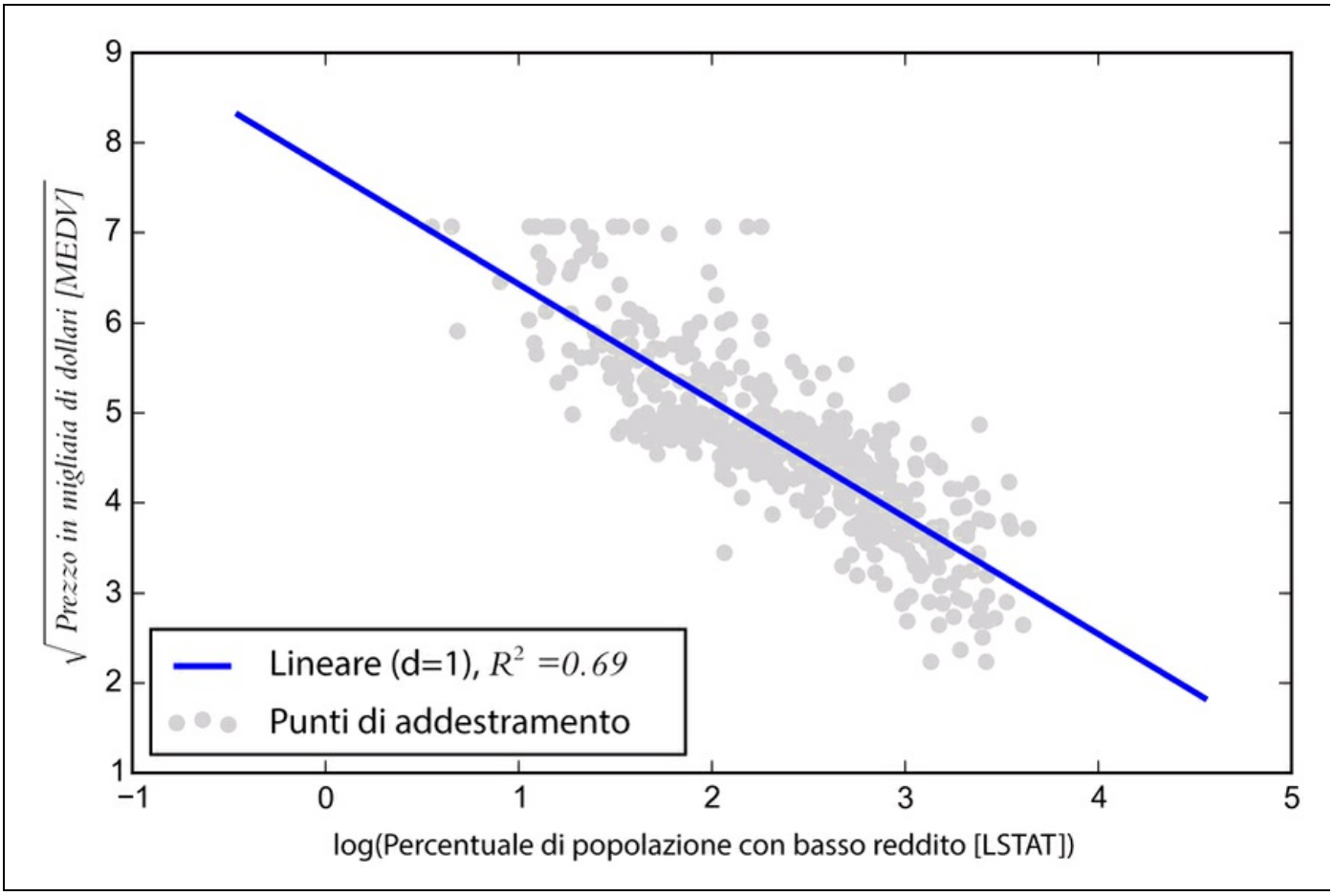


Figura 10.12

Regressione ad albero decisionale

Un vantaggio dell'algoritmo ad albero decisionale è il fatto che non richiede alcuna trasformazione delle caratteristiche se stiamo utilizzando dati non lineari. Ricordiamo dal Capitolo 3, *I classificatori di machine learning di scikit-learn*, che facciamo crescere un albero decisionale tramite una suddivisione iterativa dei suoi nodi, fino a ottenere foglie pure oppure ffino a soddisfare un determinato criterio. Quando abbiamo utilizzato gli alberi decisionali per le attività di classificazione, abbiamo definito l'entropia come una misura delle impurità, per determinare quale suddivisione delle caratteristiche massimizza il guadagno informativo (*IG – Information Gain*), che, per una suddivisione binaria, può essere definito nel seguente modo:

$$IG\left(D_p, x\right) = I\left(D_p\right) - \frac{1}{N_p} I$$

Qui, *x* è la caratteristica sulla quale eseguire la suddivisione, N_p è il numero di campioni nel nodo genitore, I è la funzione di impurità e D_p è il sottoinsieme dei campioni di addestramento nel nodo genitore. Il nostro obiettivo è quello di trovare la suddivisione della caratteristica in grado di massimizzare il guadagno informativo; in altre parole, vogliamo trovare quella suddivisione delle caratteristiche che consenta di ridurre le impurità nei nodi figli. Nel Capitolo 3, *I classificatori di machine learning di scikit-learn*, per misurare l'impurità abbiamo utilizzato l'entropia, che può essere un utile criterio di classificazione. Per utilizzare un albero decisionale per la regressione, sostituiremo l'entropia, quale misuratore delle impurità di un nodo *t*, con l'errore MSE:

$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} \left(y^{(i)} - \hat{y}_t \right)^2$$

Qui, N_t è il numero di campioni di addestramento nel nodo t, D_t è il sottoinsieme di addestramento nel nodo t, $y^{(i)}$ è il vero valore obiettivo e $\hat{y}_t$ è il valore obiettivo previsto (media del campione):

$$\hat{y}_t = \frac{1}{N} \sum_{i \in D_t} y^{(i)}$$

Nel contesto della regressione ad albero decisionale, l'errore MSE viene spesso chiamato varianza interna del nodo, motivo per cui il criterio di suddivisione è chiamato anche *riduzione della varianza*. Per vedere l'aspetto della linea di adattamento di un albero decisionale, utilizziamo il DecisionTreeRegressor implementato in scikit-learn per modellare la relazione non lineare fra le variabili MEDV e LSTAT:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
>>> tree.fit(X, y)
>>> sort_idx = X.flatten().argsort()
>>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.show()
```

Come possiamo vedere dalla Figura 10.13, l'albero decisionale cattura la tendenza generale presente nei dati. Tuttavia, un limite di questo modello è il fatto che non cattura la continuità e la differenziabilità della previsione desiderata. Inoltre, dobbiamo fare attenzione alla scelta di un valore appropriato per la

profondità dell'albero, per non incorrere in un overfitting o underfitting sui dati; qui, una profondità pari a 3 sembra essere una buona scelta.

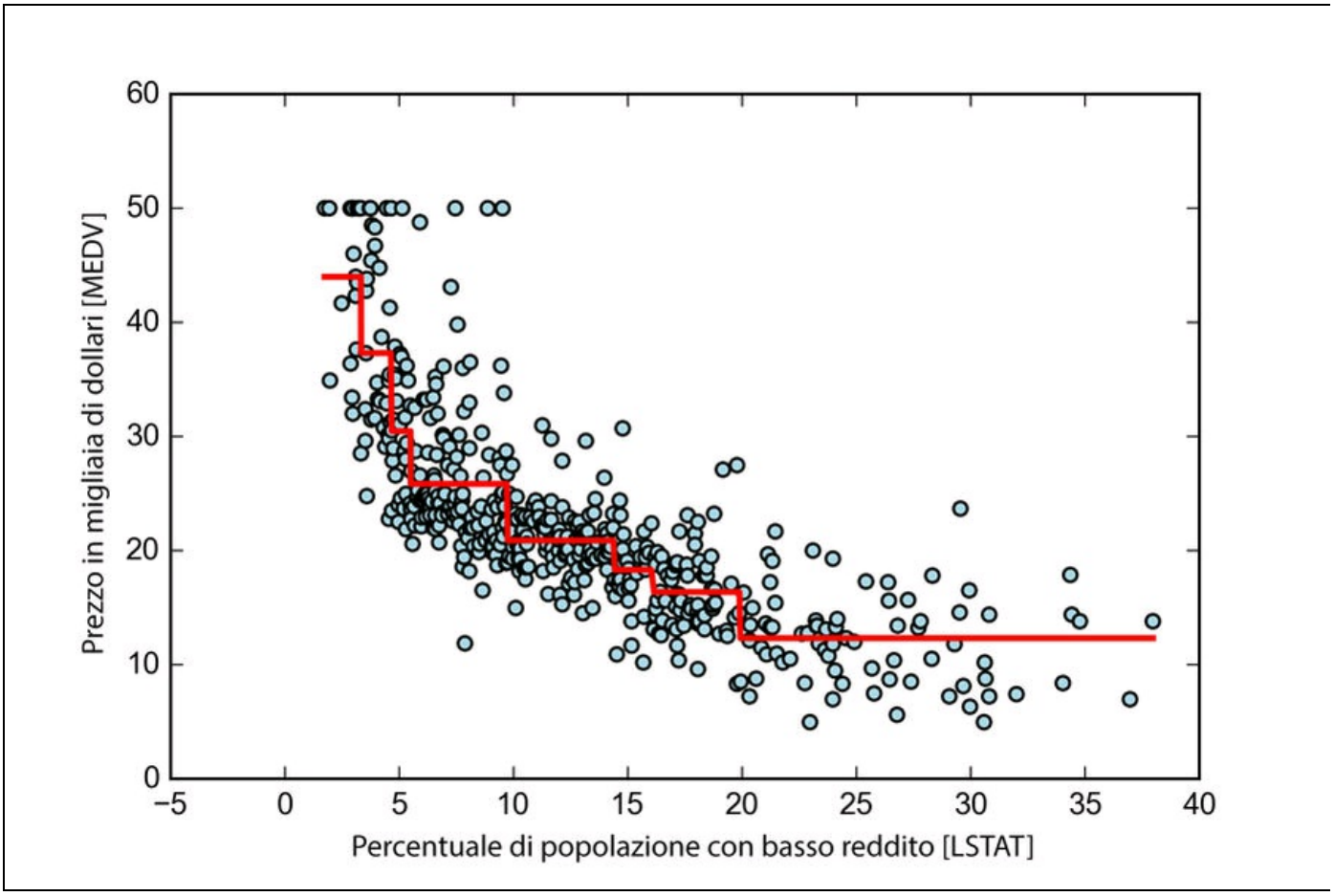


Figura 10.13

Nel prossimo paragrafo parleremo di un metodo più solido rispetto agli alberi a regressione: le foreste casuali.

Regressione a foresta casuale

Come abbiamo visto nel Capitolo 3, *I classificatori di machine learning di scikit-learn*, l'algoritmo a foresta casuale è una tecnica d'insieme che combina più alberi decisionali. Una foresta casuale, normalmente, ha prestazioni di generalizzazione migliori rispetto a un singolo albero decisionale, grazie alla casualità che aiuta a ridurre la varianza del modello. Le foreste casuali presentano anche altri vantaggi: sono meno sensibili ai valori anomali non presenti nel dataset e non richiedono una particolare ottimizzazione dei parametri. L'unico parametro delle foreste casuali con il quale dobbiamo normalmente sperimentare è il numero di alberi dell'insieme.

La versione semplice dell'algoritmo a foreste casuali per la regressione è quasi identica all'algoritmo per foreste casuali per la classificazione di cui abbiamo già

parlato nel Capitolo 3, *I classificatori di machine learning di scikit-learn*. L'unica differenza è che utilizziamo il criterio MSE per far crescere i singoli alberi decisionali e che la variabile target prevista viene calcolata in base alla previsione media di tutti gli alberi decisionali.

Ora, utilizziamo tutte le caratteristiche del dataset Housing per adattare un modello a regressione a foresta casuale su un 60% dei campioni e valutiamo le sue prestazioni sul rimanente 40% dei campioni. Il codice è il seguente:

```
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test =\
...     train_test_split(X, y,
...                      test_size=0.4,
...                      random_state=1)
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(
...                          n_estimators=1000,
...                          criterion='mse',
...                          random_state=1,
...                          n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> y_train_pred = forest.predict(X_train)
>>> y_test_pred = forest.predict(X_test)
>>> print('MSE train: %.3f, test: %.3f' % (
...     mean_squared_error(y_train, y_train_pred),
...     mean_squared_error(y_test, y_test_pred)))
>>> print('R^2 train: %.3f, test: %.3f' % (
...     r2_score(y_train, y_train_pred),
...     r2_score(y_test, y_test_pred)))
MSE train: 1.642, test: 11.635
R^2 train: 0.960, test: 0.871
```

Sfortunatamente, vediamo che la foresta casuale tende a manifestare un overfitting sui dati di addestramento. Tuttavia, è comunque in grado di individuare in modo non trascurabile la relazione esistente fra le variabili target e descrittive ($R^2 = 0.871$ sul dataset di test).

Infine, diamo un'occhiata anche ai residui della previsione:

```
>>> plt.scatter(y_train_pred,
...            y_train_pred - y_train,
...            c='black',
...            marker='o',
...            s=35,
...            alpha=0.5,
...            label='Training data')
>>> plt.scatter(y_test_pred,
...            y_test_pred - y_test,
...            c='lightgreen',
...            marker='s',
...            s=35,
...            alpha=0.7,
...            label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

Come abbiamo già riepilogato per il coefficiente R^2, possiamo vedere che il modello si adatta ai dati di addestramento meglio rispetto ai dati di test, come indicato dai valori anomali nella direzione dell'asse *y*. Inoltre, la distribuzione dei

residui non sembra essere completamente casuale attorno al punto centrale 0 e ciò indica il fatto che il modello non è in grado di catturare tutte le informazioni descrittive. Tuttavia, il grafico dei residui (Figura 10.14) indica un grande miglioramento rispetto al grafico dei residui del modello lineare che abbiamo tracciato in precedenza in questo stesso capitolo.

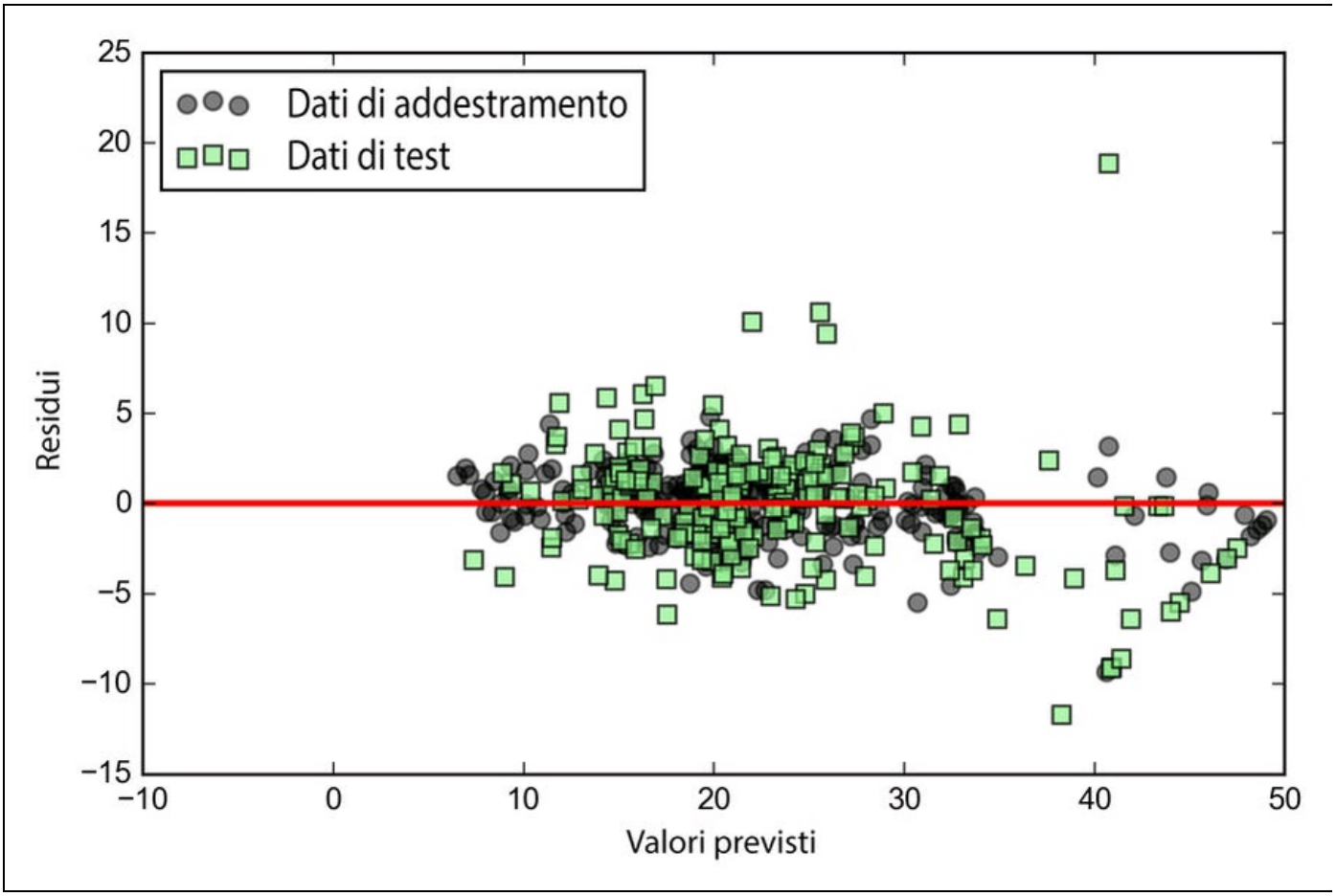


Figura 10.14

NOTA

Nel Capitolo 3, *I classificatori di machine learning di scikit-learn*, abbiamo anche introdotto una tecnica che può essere utilizzata per la classificazione in combinazione con la macchina a vettori di supporto (*SVM – support vector machine*), il che è utile se dobbiamo occuparci di problemi non lineari. Sebbene una discussione più approfondita non rientri negli scopi di questo libro, le macchine SVM possono essere utilizzate anche in compiti di regressione non lineare. Il lettore interessato può trovare ulteriori informazioni sull'uso di macchine SVM per la regressione in un eccellente opera di S. R. Gunn: S. R. Gunn et al, *Support Vector Machines for Classification and Regression*, in "ISIS technical report", 14, 1998. Un regressore SVM è implementato anche in scikit-learn; ulteriori informazioni sul suo uso si trovano in http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR.

Riepilogo

All'inizio di questo capitolo, abbiamo parlato dell'uso dell'analisi a regressione lineare semplice per modellare la relazione esistente fra un'unica variabile descrittiva e una variabile a risposta continua. Abbiamo poi esaminato una tecnica utile per l'analisi dei dati descrittivi, alla ricerca di schemi e anomalie nei dati, il che può essere un primo passo importante nei compiti di modellazione previsionale.

Abbiamo costruito il nostro primo modello implementando la regressione lineare tramite un approccio di ottimizzazione basato sul gradiente. Abbiamo poi visto come utilizzare i modelli lineari di scikit-learn per la regressione e anche come implementare una solida tecnica di regressione (RANSAC) come un approccio per risolvere il problema dei valori anomali. Per valutare le prestazioni dei modelli a regressione, abbiamo calcolato gli errori e la relativa metrica R^2. Inoltre, abbiamo esaminato un utile approccio grafico per diagnosticare i problemi dei modelli a regressione: il grafico dei residui.

Dopo aver discusso di come applicare la regolarizzazione ai modelli a regressione, con lo scopo di ridurre la complessità del modello ed evitare il problema dell'overfitting, abbiamo anche introdotto vari approcci alle relazioni non lineari, fra cui la trasformazione polinomiale delle caratteristiche e i regressori a foresta casuale.

In questi capitoli abbiamo parlato piuttosto in dettaglio di apprendimento con supervisione, classificazione e analisi a regressione. Nel prossimo capitolo parleremo di un altra branca interessante del machine learning: l'apprendimento senza supervisione. Nel prossimo capitolo impareremo a utilizzare l'analisi a cluster per scoprire le strutture nascoste nei dati, in assenza di variabili target.

Lavorare con dati senza etichette: l'analisi a cluster

Nei capitoli precedenti, abbiamo utilizzato tecniche di apprendimento con supervisione, realizzando quindi modelli di apprendimento che utilizzavano dati per i quali la risposta era già nota: le etichette delle classi erano già disponibili nei dati di addestramento. In questo capitolo cambieremo marcia ed esploreremo l'analisi a cluster, una categoria di tecniche di *apprendimento senza supervisione*, il quale ci consente di individuare le strutture nascoste nei dati, informazioni per le quali non conosciamo la risposta corretta prima di iniziare. L'obiettivo del clustering è quello di individuare un raggruppamento naturale presente nei dati: trovare quel "cluster" di elementi che sono più simili fra loro rispetto a quelli che rientrano in altri cluster.

Data la sua natura esplorativa, il clustering è un argomento davvero avvincente e, in questo capitolo, tratteremo i seguenti concetti, che ci aiuteranno a organizzare i dati in strutture significative.

- Trovare i centri di similarità utilizzando il popolare algoritmo k-means.
- Utilizzare un approccio bottom-up per realizzare alberi gerarchici di cluster.
- Identificare forme arbitrarie degli oggetti, utilizzando un approccio a clustering basato sulla densità.

Raggruppare gli oggetti per similarità utilizzando l'algoritmo k-means

In questo paragrafo parleremo di uno dei più noti algoritmi di clustering, *k-means*, ampiamente utilizzato in ambito accademico e produttivo. Il *clustering* (o *analisi a cluster*) è una tecnica che consente di individuare gruppi di oggetti simili, oggetti che sono maggiormente correlati fra loro rispetto a quelli appartenenti ad altri gruppi. Fra gli esempi di applicazioni commerciali del clustering vi è il raggruppamento di documenti, di brani musicali e di film sulla base di vari argomenti o la ricerca di quei clienti che condividono determinati interessi sulla base delle loro precedenti abitudini di acquisto, da impiegare per inviare loro suggerimenti commerciali da parte delle engine dedicate.

Come vedremo fra poco, l'algoritmo k-means è estremamente facile da implementare, ma è anche molto efficiente dal punto di vista computazionale rispetto ad altri algoritmi di clustering e questo spiega la sua popolarità. L'algoritmo k-means appartiene alla categoria del clustering basato su prototipi. Parleremo delle altre due categorie del clustering, *gerarchico* e *basato su densità*, più avanti in questo stesso capitolo. Con *clustering basato su prototipi* si intende il fatto che ogni cluster è rappresentato da un prototipo, che può essere il *centroide* (la media), di punti simili e con caratteristiche continue, oppure il *medoide* (il punto più rappresentativo o che si presenta più frequentemente) nel caso di caratteristiche che si presentano suddivise in categorie (categoriche). Mentre k-means è molto efficace nell'identificare i cluster di forma sferica, uno dei suoi difetti è il fatto che richiede di specificare a priori il numero di cluster, *k*. Una scelta non appropriata di *k* può produrre cattive prestazioni in termini di clustering. Più avanti in questo capitolo, parleremo del metodo *Elbow* (letteralmente "gomito") e dei grafici a *silhouette*, tecniche utili per valutare la qualità del clustering, con lo scopo di determinare il numero ottimale dei cluster *k*.

Sebbene il clustering k-means possa essere applicato a dati di dimensioni più elevate, svolgeremo i seguenti esempi utilizzando un semplice dataset bidimensionale, più facile da rappresentare:

```
>>> from sklearn.datasets import make_blobs
>>> X, y = make_blobs(n_samples=150,
...                   n_features=2,
...                   centers=3,
...                   cluster_std=0.5,
...                   shuffle=True,
...                   random_state=0)
```

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter(X[:,0],
...             X[:,1],
...             c='white',
...             marker='o',
...             s=50)
>>> plt.grid()
>>> plt.show()
```

Il dataset che abbiamo appena creato è costituito da 150 punti generati casualmente, che si trovano approssimativamente raggruppati in tre regioni che presentano densità più elevate, come si può vedere nella Figura 11.1.

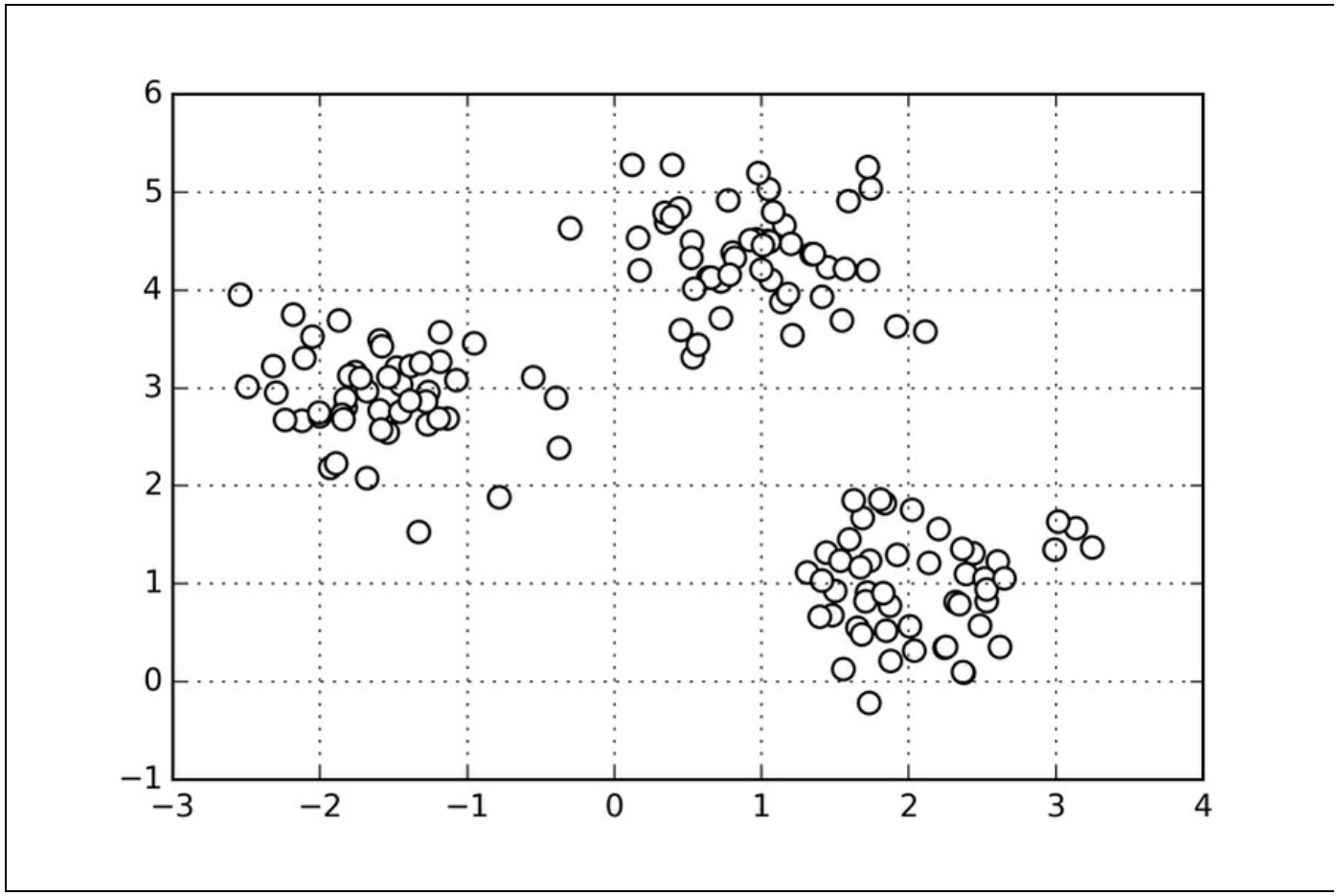


Figura 11.1

Nelle applicazioni del mondo reale, quando si parla di clustering, in realtà non abbiamo a disposizione informazioni particolarmente utili su questi campioni; altrimenti tutto rientrerebbe nella categoria dell'apprendimento con supervisione. Pertanto, il nostro obiettivo consiste nel raggruppare i campioni sulla base delle similarità individuate fra le loro caratteristiche; questo può essere ottenuto utilizzando l'algoritmo k-means, che può essere riepilogato nei seguenti quattro passi.

1. Selezionare casualmente *k* centroidi dei punti campione; essi fungeranno da centri provvisori dei cluster.

2. Assegnare a ciascun campione il centroide più vicino $\mu^{(j)}$, $j \in \{1, \ldots, k\}$
3. Spostare centroidi al centro dei campioni che gli sono stati assegnati.
4. Ripetere i Passi 2 e 3 finché l'assegnamento dei cluster smette di cambiare oppure quando viene raggiunta una determinata tolleranza o un determinato numero massimo di iterazioni.

Ora, la domanda successiva è *come misurare la similarità fra gli oggetti*? Possiamo definire la similarità come l'opposto della distanza e una distanza comunemente utilizzata per il clustering di campioni dotati di caratteristiche continue è la *distanza euclidea al quadrato* fra *x* e *y* in uno spazio *m*-dimensionale:

$$d(\boldsymbol{x}, \boldsymbol{y})^2 = \sum_{j=1}^{m} \left(x_j - y_j\right)^2 = \left\|\boldsymbol{x} - \boldsymbol{y}\right\|_2^2$$

Notate che, nell'equazione precedente, l'indice *j* fa riferimento alla *j*-esima dimensione (colonna delle caratteristiche) dei punti campione *x* e *y*. Nel resto di questo paragrafo, utilizzeremo gli indici *i* e *j* per far riferimento, rispettivamente, all'indice del campione e all'indice del cluster.

Sulla base di questa distanza metrica euclidea, possiamo descrivere l'algoritmo k-means come un semplice problema di ottimizzazione, un approccio iterativo per minimizzare il valore *SSE* (*sum of squared errors*, ovvero la somma degli errori quadratici all'interno del cluster), che viene anche chiamata *inerzia del cluster*:

$$SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)} \left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)}\right\|_2^2$$

Qui, $\mu^{(j)}$ è il punto rappresentativo (centroide) del cluster *j* e $w^{(i,j)} = 1$ se il campione $\boldsymbol{x}^{(i)}$ è nel cluster *j*, altrimenti è $w^{(i,j)} = 0$.

Ora che conosciamo le basi del funzionamento dell'algoritmo k-means, applichiamolo al nostro dataset campione utilizzando la classe `KMeans` del modulo `cluster` di scikit-learn:

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
...             init='random',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
```

```
...             random_state=0)
>>> y_km = km.fit_predict(X)
```

Utilizzando il codice precedente, impostiamo il numero di cluster desiderati a `3`; il fatto di dover specificare a priori il numero dei cluster è uno dei limiti di k-means. Impostiamo `n_init=10`, per eseguire gli algoritmi di clustering k-means per 10 volte, in modo indipendente con centroidi casuali differenti, per scegliere il modello finale che offre il valore SSE più basso. Tramite il parametro `max_iter`, specifichiamo il numero massimo di iterazioni per ciascuna esecuzione (in questo caso 300). Notate che l'implementazione di k-means in scikit-learn è in grado di fermarsi anche prima, se si accorge che riesce a convergere prima di aver raggiunto il numero massimo di iterazioni.

Tuttavia, è possibile che k-means non raggiunga la convergenza entro una determinata esecuzione, il che può essere un problema (costoso dal punto di vista computazionale) se scegliamo valori relativamente grandi di `max_iter`. Un modo per risolvere i problemi di convergenza consiste nello scegliere valori piuttosto elevati per `tol`, un parametro che controlla la tolleranza rispetto alle variazioni nella somma degli errori al quadrato all'interno del cluster tali da dichiarare la convergenza. Nel codice precedente, abbiamo scelto una tolleranza pari a `1e-04` (=0.0001).

K-means++

Finora, abbiamo parlato dell'algoritmo k-means classico, che utilizza un seme casuale per collocare i centroidi iniziali, il che può talvolta produrre un cattivo clustering o una lenta convergenza nel caso in cui tali centroidi fossero scelti in modo poco appropriato. Un modo per risolvere questo problema consiste nell'eseguire l'algoritmo k-means più volte sul dataset e scegliere il risultato migliore in termini di SSE. Un'altra strategia consiste nel collocare i centroidi iniziali ben distanziati fra loro, impiegando l'algoritmo *k-means++*, che porta a risultati migliori e più coerenti rispetto al classico k-means (D. Arthur e S. Vassilvitskii, *k-means++: The Advantages of Careful Seeding*, in "Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms", pp. 1027–1035. Society for Industrial and Applied Mathematics, 2007).

L'inizializzazione dell'algoritmo k-means++ può essere riepilogata nel seguente modo.

1. Inizializzare un insieme vuoto $\mathbf{M}$ che conterrà i k centroidi selezionati.

2. Scegliere casualmente il primo centroide $\mu^{(j)}$ dai campioni di input e assegnarlo a $\mathbf{M}$.
3. Per ciascun campione $x^{(i)}$ che non è presente in $\mathbf{M}$, trovare la distanza quadratica minima

   $$d\left(x^{(i)}, \mathbf{M}\right)^2$$

   verso i centroidi $\mathbf{M}$.
4. Per scegliere casualmente il centroide successivo $\mu^{(p)}$, utilizzare una distribuzione di probabilità pesata, uguale a:

   $$\frac{d\left(\mu^{(p)}, \mathbf{M}\right)^2}{\sum_i d\left(x^{(i)}, \mathbf{M}\right)^2}$$
5. Ripetere i Passi 2 e 3 fino a scegliere i $k$centroidi.
6. Procedere poi con l'algoritmo k-means classico.

NOTA

Per utilizzare k-means++ con l'oggetto KMeans di scikit-learn, abbiamo solo bisogno di impostare il parametro init a k-means++ (l'impostazione predefinita) invece di utilizzare random.

Un altro problema di k-means è il fatto che uno o più cluster possono essere vuoti. Notate che questo problema non esiste per le tecniche k-medoids (o fuzzy C-means), un algoritmo di cui parleremo nel prossimo paragrafo. Tuttavia, questo problema è presente nell'attuale implementazione di k-means presente in scikit-learn. Se un cluster è vuoto, l'algoritmo ricercherà il campione più lontano dal centroide del cluster vuoto. Poi ricollocherà il centroide su questo punto lontano.

NOTA

Quando applichiamo l'algoritmo k-means ai dati prodotti nel mondo reale utilizzando una metrica di distanza euclidea, vogliamo assicurarci che le caratteristiche vengano misurate sulla stessa scala, applicando, se necessario, la standardizzazione z-score o la riduzione in scala min-max.

Dopo aver previsto le etichette y_km dei cluster e aver discusso i problemi dell'algoritmo k-means, vediamo come vengono identificati i cluster da k-means nel dataset, insieme ai centroidi del cluster. Questi vengono memorizzati sotto l'attributo centers_ dell'oggetto KMeans:

```
>>> plt.scatter(X[y_km==0,0],
...             X[y_km==0,1],
...             s=50,
...             c='lightgreen',
```

```
...             marker='s',
...             label='cluster 1')
>>> plt.scatter(X[y_km==1,0],
...             X[y_km==1,1],
...             s=50,
...             c='orange',
...             marker='o',
...             label='cluster 2')
>>> plt.scatter(X[y_km==2,0],
...             X[y_km==2,1],
...             s=50,
...             c='lightblue',
...             marker='v',
...             label='cluster 3')
>>> plt.scatter(km.cluster_centers_[:,0],
...             km.cluster_centers_[:,1],
...             s=250,
...             marker='*',
...             c='red',
...             label='centroids')
>>> plt.legend()
>>> plt.grid()
>>> plt.show()
```

Nel grafico rappresentato nella Figura 11.2, possiamo vedere che k-means ha collocato i tre centroidi al centro di ciascuna "sfera" di punti, una scelta che sembra ragionevole, osservando questo dataset.

Sebbene k-means abbia funzionato bene su questo semplice dataset, dobbiamo notare alcuni dei suoi principali difetti. Il primo è che occorre specificare a priori il numero dei cluster *k*, il che non sempre può essere così ovvio nelle applicazioni pratiche, specialmente quando si lavora con un dataset a elevata dimensionalità e che non può essere visualizzato. Le altre proprietà di k-means sono il fatto che i cluster non si sovrappongano e non siano gerarchici; inoltre si presuppone che ogni cluster contenga almeno un elemento.

Clustering hard e soft

Con *clustering hard* si definisce una famiglia di algoritmi nella quale ciascun campione di un dataset viene assegnato esattamente a un cluster, come nell'algoritmo k-means di cui abbiamo parlato nel paragrafo precedente. Al contrario, gli algoritmi di *clustering soft* (o *fuzzy*) assegnano un campione a uno o più cluster. Un tipico esempio di clustering soft è l'algoritmo *FCM* (fuzzy C-means), chiamato anche *soft k-means* o *fuzzy k-means*. L'idea originale risale agli anni Settanta, quando Joseph C. Dunn propose una versione preliminare del clustering fuzzy per migliorare k-means (J. C. Dunn, *A Fuzzy Relative of the Isodata Process and its Use in Detecting Compact Well-separated Clusters*, 1973). Quasi un decennio dopo, James C. Bedzek ha pubblicato il proprio lavoro sui miglioramenti dell'algoritmo di clustering fuzzy, che oggi vanno sotto il nome di

algoritmo FCM (J. C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms*, Springer Science & Business Media, 2013).

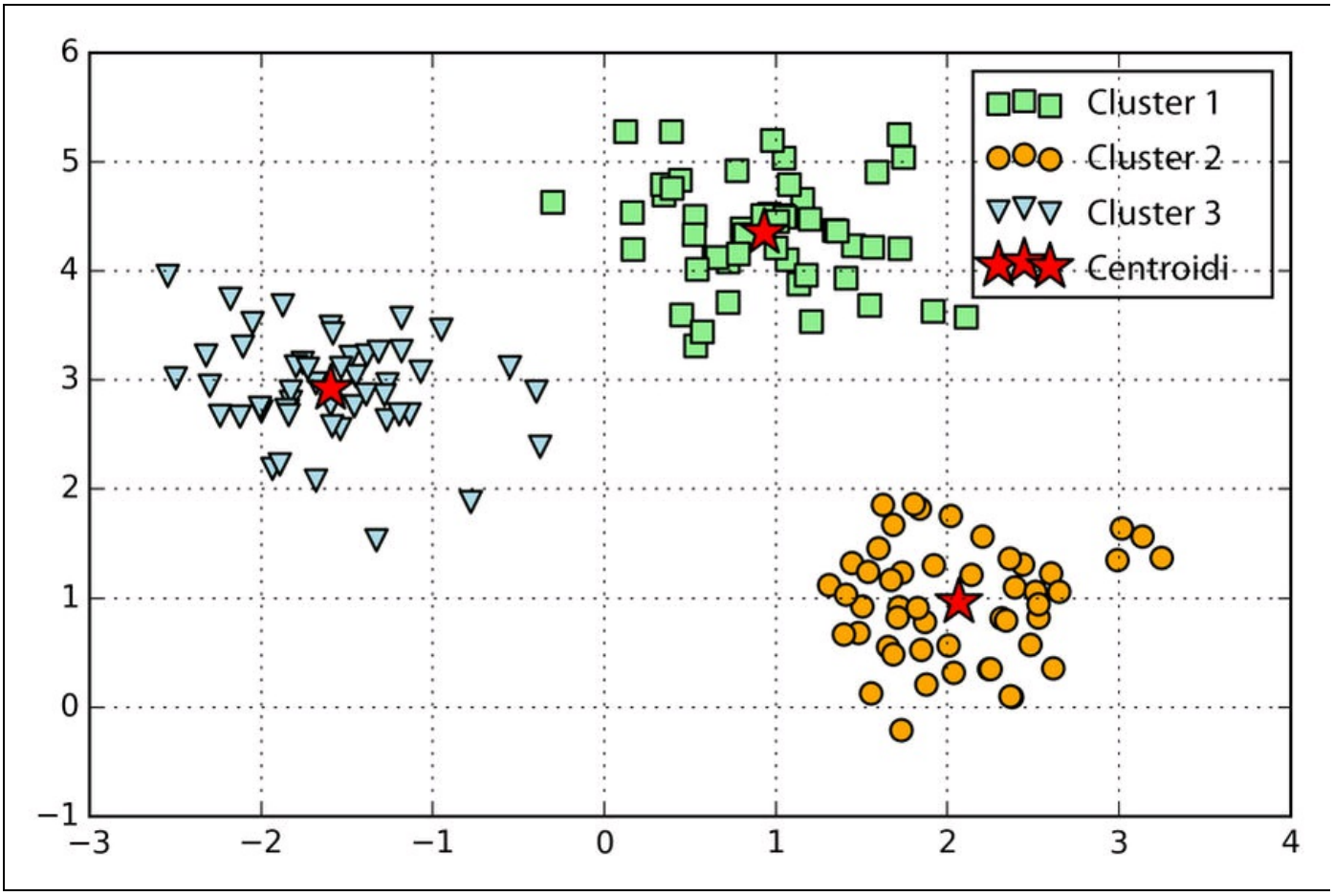


Figura 11.2

Il funzionamento dell'algoritmo FCM è molto simile a quello di k-means. Tuttavia, sostituiamo l'assegnamento specifico a ciascun cluster con le probabilità che ogni punto appartenga a un cluster. In k-means, esprimeremo l'appartenenza di un campione *x* a un cluster tramite un vettore sparso di valori binari:

$$\begin{bmatrix} \mu^{(1)} \rightarrow 0 \\ \mu^{(2)} \rightarrow 1 \\ \mu^{(3)} \rightarrow 0 \end{bmatrix}$$

Qui, la posizione indice con valore 1 indica il centroide del cluster $\mu^{(j)}$ al quale viene assegnato il campione (supponendo $k = 3,\ j \in \{1, 2, 3\}$). Al contrario, in FCM un vettore di appartenenza può essere rappresentato nel seguente modo:

$$\begin{bmatrix} \boldsymbol{\mu}^{(1)} \rightarrow 0.1 \\ \boldsymbol{\mu}^{(2)} \rightarrow 0.85 \\ \boldsymbol{\mu}^{(3)} \rightarrow 0.05 \end{bmatrix}$$

Qui, ciascun valore rientra nell'intervallo [0, 1] e rappresenta una probabilità di appartenenza al rispettivo centroide di un cluster. La somma delle appartenenze per un determinato campione è sempre uguale a 1. Analogamente a quanto avviene con l'algoritmo k-means, possiamo riepilogare l'algoritmo FCM in quattro passi chiave.

1. Specificare il numero dei *k* centroidi e assegnare casualmente l'appartenenza dei cluster per ciascun punto.
2. Calcolare i centroidi dei cluster $\boldsymbol{\mu}^{(j)}$, $j \in \{1, \ldots, k\}$
3. Aggiornare l'appartenenza ai cluster per ciascun punto.
4. Ripetere i Passi 2 e 3 finché i coefficienti di appartenenza non cambiano o finché non viene raggiunta una specifica tolleranza definita dall'utente o viene svolto un numero massimo di iterazioni.

La funzione obiettivo di FCM (che abbreviamo con J_m) ha un aspetto molto simile alla funzione *sum-squared-error* che dobbiamo minimizzare con l'algoritmo k-means:

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{m\,(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2 , \; m \in [1, \infty)$$

Tuttavia, notate che l'indicatore di appartenenza $w^{(i,j)}$ non è un valore binario come in k-means

$$w^{(i,j)} \in \{0, 1\}$$

ma un valore reale, che denota la probabilità di appartenenza al cluster

$$w^{(i,j)} \in [0, 1]$$

)

Potete anche aver notato che abbiamo aggiunto a $w^{(i,j)}$ un esponente in più. L'esponente *m*, un numero maggiore o uguale a 1 (in genere *m* = *2*), è il cosiddetto coefficiente di *fuzziness* (o *fuzzifier*). Maggiore è il valore di *m*, più piccola diviene l'appartenenza al cluster $w^{(i,j)}$, che porta a cluster più "confusi". La probabilità di appartenenza ai cluster viene invece calcolata nel seguente modo:

$$w^{(i,j)} = \left[\sum_{p=1}^{k} \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(p)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

Per esempio, se scegliessimo tre centri per i cluster, come nel precedente esempio con k-means, potremmo calcolare l'appartenenza del campione $\boldsymbol{x}^{(i)}$ appartenente al cluster $\boldsymbol{\mu}^{(j)}$ nel seguente modo:

$$w^{(i,j)} = \left[\left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

Il centro $\boldsymbol{\mu}^{(j)}$ di un cluster viene calcolato come la media di tutti i campioni del cluster, pesata dal grado di appartenenza al relativo cluster:

$$\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^{n} w^{m(i,j)} \boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} w^{m(i,j)}}$$

Semplicemente osservando l'equazione necessaria per calcolare l'appartenenza al cluster, è intuitivo pensare che ogni iterazione di FCM sia più costosa rispetto a un'iterazione di k-means. Tuttavia, in genere, in media FCM richiede meno iterazioni per raggiungere la convergenza. Sfortunatamente, l'algoritmo FCM non è attualmente implementato in scikit-learn. Tuttavia, nella pratica si è trovato che sia k-means, sia FCM producono schemi di clustering molto simili, come descritto in uno studio di Soumi Ghosh e Sanjay K. Dubey (S. Ghosh e S. K. Dubey,

Comparative Analysis of k-means and Fuzzy c-means Algorithms, in "IJACSA", 4:35–38, 2013).

Il metodo Elbow per trovare il numero ottimale di cluster

Una delle sfide principali, nei casi di apprendimento senza supervisione, è il fatto che non si conosce alcuna risposta definitiva. Non abbiamo la possibilità di contare su etichette delle classi già verificate e presenti nel dataset, che ci consentano di applicare le tecniche che abbiamo utilizzato nel Capitolo 6, *Valutazione dei modelli e ottimizzazione degli iperparametri*, per valutare le prestazioni di un modello con supervisione. Pertanto, per poter quantificare la qualità del clustering, abbiamo bisogno di impiegare una metrica intrinseca (ne è un esempio l'errore SSE interno del cluster di cui abbiamo parlato in precedenza in questo capitolo) per confrontare le prestazioni dei vari esempi di clustering ottenuti tramite k-means. Fortunatamente non abbiamo bisogno di calcolare esplicitamente il valore SSE, in quanto risulta già accessibile tramite l'attributo `inertia_` dopo aver adattato un modello `KMeans`:

```
>>> print('Distortion: %.2f' % km.inertia_)
Distortion: 72.48
```

Sulla base del valore SSE, possiamo utilizzare uno strumento grafico, il cosiddetto metodo Elbow per stimare il numero ottimale di cluster *k* per un determinato compito. Intuitivamente, possiamo dire che aumentando *k* si riduce la distorsione. Questo perché i campioni saranno più vicini ai centroidi ai quali sono stati assegnati. L'idea su cui si basa il metodo Elbow è quella di identificare il valore di *k* in cui la distorsione comincia a crescere più rapidamente, un argomento che diventerà più chiaro tracciando il grafico della distorsione per più valori di *k* (che presenterà così un tracciato a "gomito piegato", NdR):

```
>>> distortions = []
>>> for i in range(1, 11):
...     km = KMeans(n_clusters=i,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 random_state=0)
>>>     km.fit(X)
>>>     distortions.append(km.inertia_)
>>> plt.plot(range(1,11), distortions, marker='o')
>>> plt.xlabel('Number of clusters')
>>> plt.ylabel('Distortion')
>>> plt.show()
```

Come possiamo vedere nella Figura 11.3, il gomito è situato in $k = 3$, il che dimostra che 3 rappresenta una buona scelta per *k* per questo dataset:

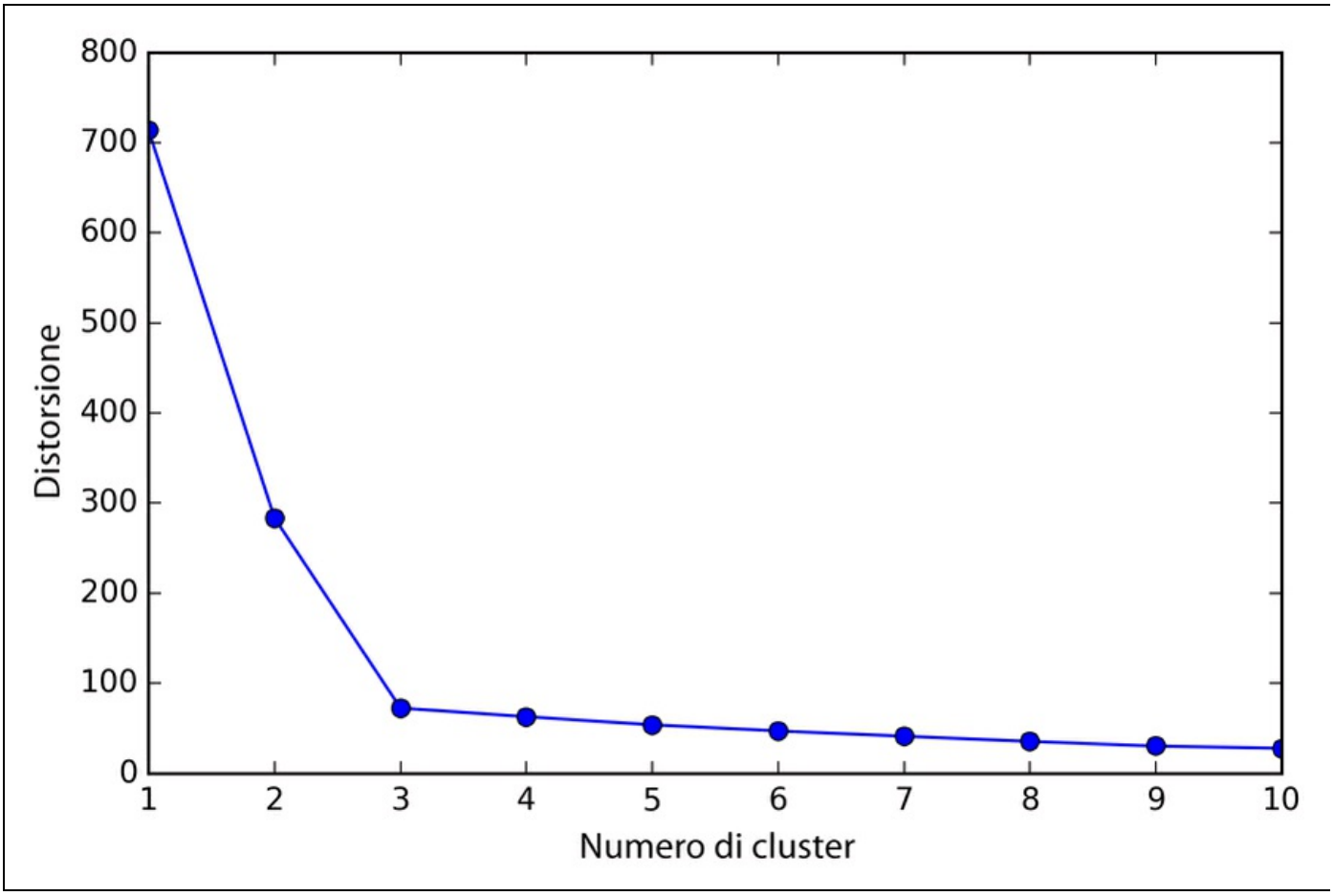


Figura 11.3

Quantificare la qualità del clustering tramite grafici a silhouettes

Un'altra metrica intrinseca per valutare la qualità di un clustering è l'*analisi della silhouette*, che può essere applicata anche ad algoritmi di clustering diversi da k-means, che esamineremo più avanti in questo capitolo. L'analisi della silhouette può essere utilizzata come strumento grafico per tracciare una misura della qualità con cui i campioni si raggruppano all'interno dei cluster. Per calcolare il coefficiente di silhouette di un singolo campione del dataset, si possono applicare i tre passi seguenti.

1. Calcolare la *coesione* del cluster $a^{(i)}$ come la distanza media fra un campione $\boldsymbol{x}^{(i)}$ e tutti gli altri punti dello stesso cluster.
2. Calcolare la *separazione* del cluster $b^{(i)}$ dal cluster più vicino come la distanza media fra il campione $\boldsymbol{x}^{(i)}$ e tutti i campioni del cluster più vicino.

3. Calcolare la silhouette $s^{(i)}$ come la differenza fra la coesione e la separazione del cluster, il tutto diviso per il maggiore dei due, come si può vedere dalla seguente formula:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\left\{b^{(i)}, a^{(i)}\right\}}$$

Il coefficiente di silhouette è limitato dall'intervallo da -1 a 1. Sulla base della formula precedente, possiamo vedere che il coefficiente di silhouette è pari a 0 se la separazione e la coesione del cluster sono uguali ($b^{(i)} = a^{(i)}$). Inoltre, possiamo avvicinarci a un coefficiente di silhouette ideale (1) se $b^{(i)} >> a^{(i)}$, in quanto $b^{(i)}$ quantifica quanto è dissimile un campione rispetto agli altri cluster e $a^{(i)}$ ci dice quanto è simile agli altri campioni del proprio cluster.

Il coefficiente di silhouette è disponibile come `silhouette_samples` dal modulo `metric` di scikit-learn e, opzionalmente, può essere importato `silhouette_scores`. Ciò calcola il coefficiente di silhouette medio fra tutti i campioni, che è equivalente a `numpy.mean(silhouette_samples(…))`. Tramite il codice seguente, tracciamo un grafico dei coefficienti di silhouette per un clustering k-means con $k = 3$:

```
>>> km = KMeans(n_clusters=3,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                      y_km,
...                                      metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(i / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...             color="red",
...             linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
```

```
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.show()
```

Tramite un'ispezione visuale del grafico (Figura 11.4), possiamo rappresentare rapidamente le dimensioni dei singoli cluster e identificare i cluster che contengono valori anomali.

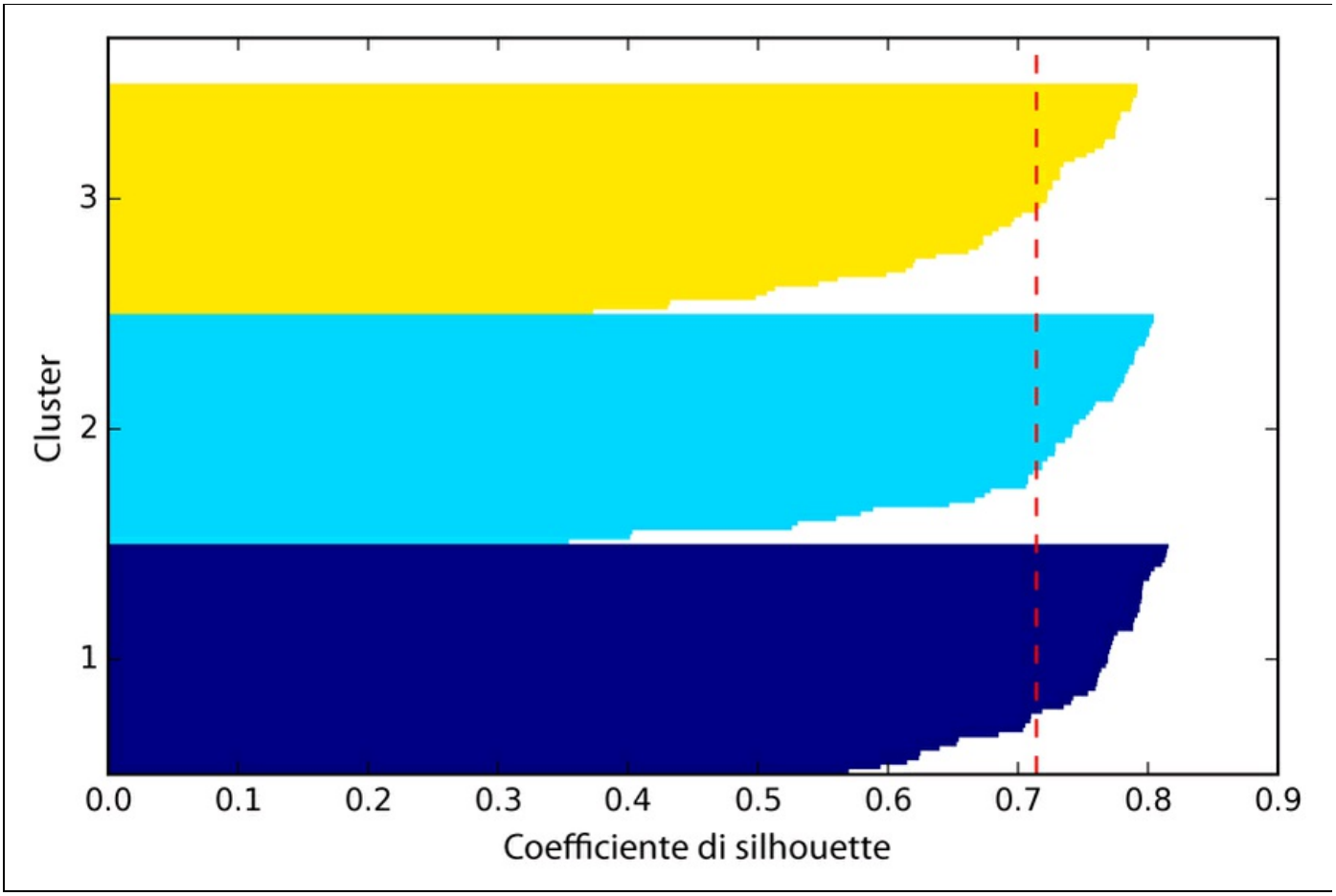


Figura 11.4

Come possiamo vedere nel grafico, i nostri coefficienti di silhouette si mantengono ben lontani da 0, il che è un indicatore di un buon clustering. Inoltre, per confrontare la bontà del clustering, abbiamo aggiunto al grafico il coefficiente di silhouette medio (la linea tratteggiata).

Per vedere l'aspetto di un grafico di silhouette nel caso di un *cattivo* esempio di clustering, applichiamo l'algoritmo di una ricerca con due soli centroidi:

```
>>> km = KMeans(n_clusters=2,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> plt.scatter(X[y_km==0,0],
...             X[y_km==0,1],
...             s=50, c='lightgreen',
...             marker='s',
...             label='cluster 1')
>>> plt.scatter(X[y_km==1,0],
```

```
...             X[y_km==1,1],
...             s=50,
...             c='orange',
...             marker='o',
...             label='cluster 2')
>>> plt.scatter(km.cluster_centers_[:,0],
...             km.cluster_centers_[:,1],
...             s=250,
...             marker='*',
...             c='red',
...             label='centroids')
>>> plt.legend()
>>> plt.grid()
>>> plt.show()
```

Come possiamo vedere nella Figura 11.5, uno dei centroidi si colloca a metà strada fra i due raggruppamenti dei punti campione. Anche se il clustering non è del tutto inappropriato, non è certo ottimale.

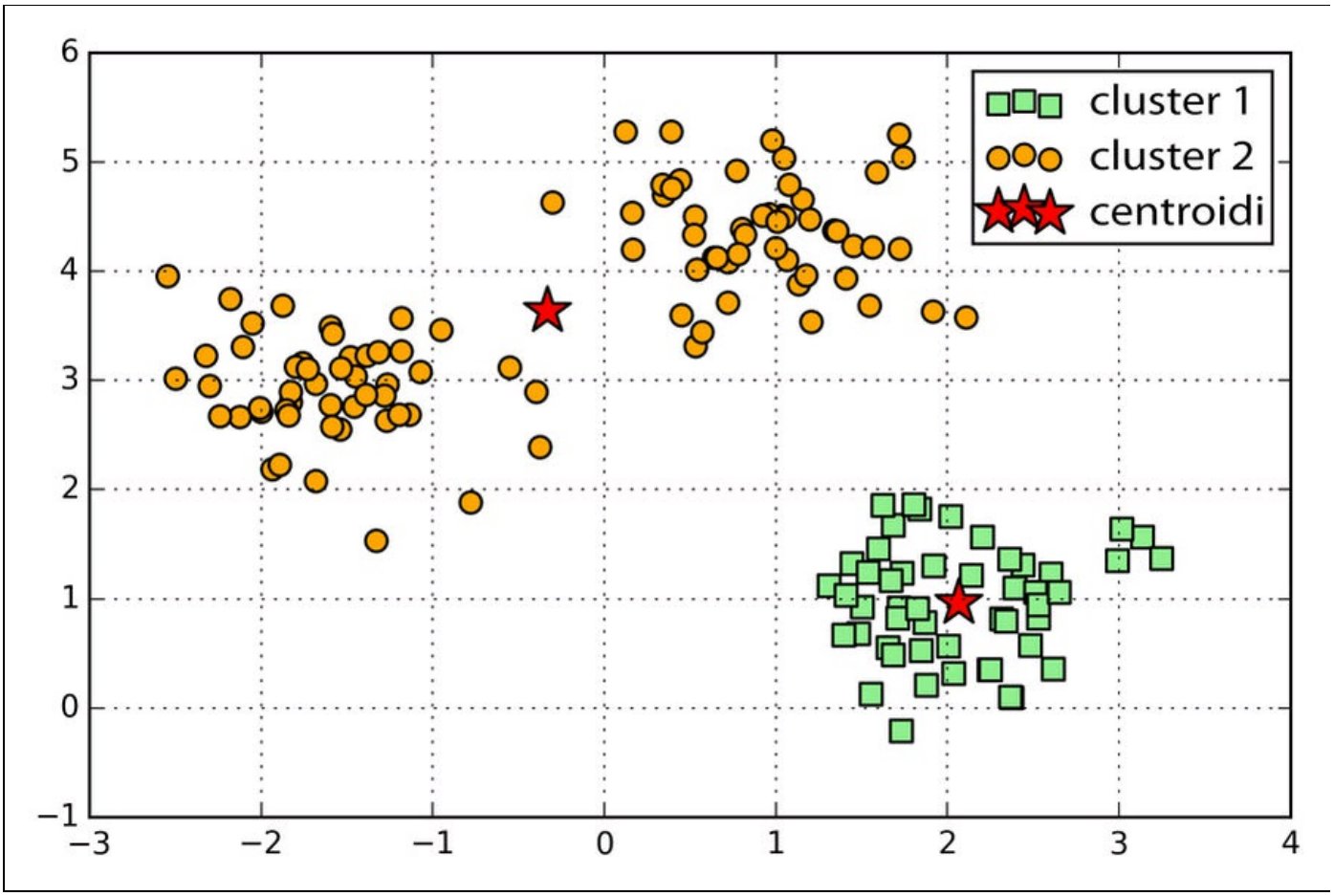


Figura 11.5

Poi creiamo il grafico delle silhouette per valutare i risultati. Tenete in considerazione che, in genere, quando abbiamo a che fare con problemi reali non abbiamo il lusso di poter rappresentare i dataset in un grafico bidimensionale, in quanto in genere operiamo su dati che hanno molte dimensioni in più:

```
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                      y_km,
...                                      metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
```

```
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(i / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.show()
```

Come si può vedere dal grafico presentato nella Figura 11.6, le silhouette ora hanno lunghezze e larghezze ben differenti, il che evidenzia il fatto che il clustering non è ottimale.

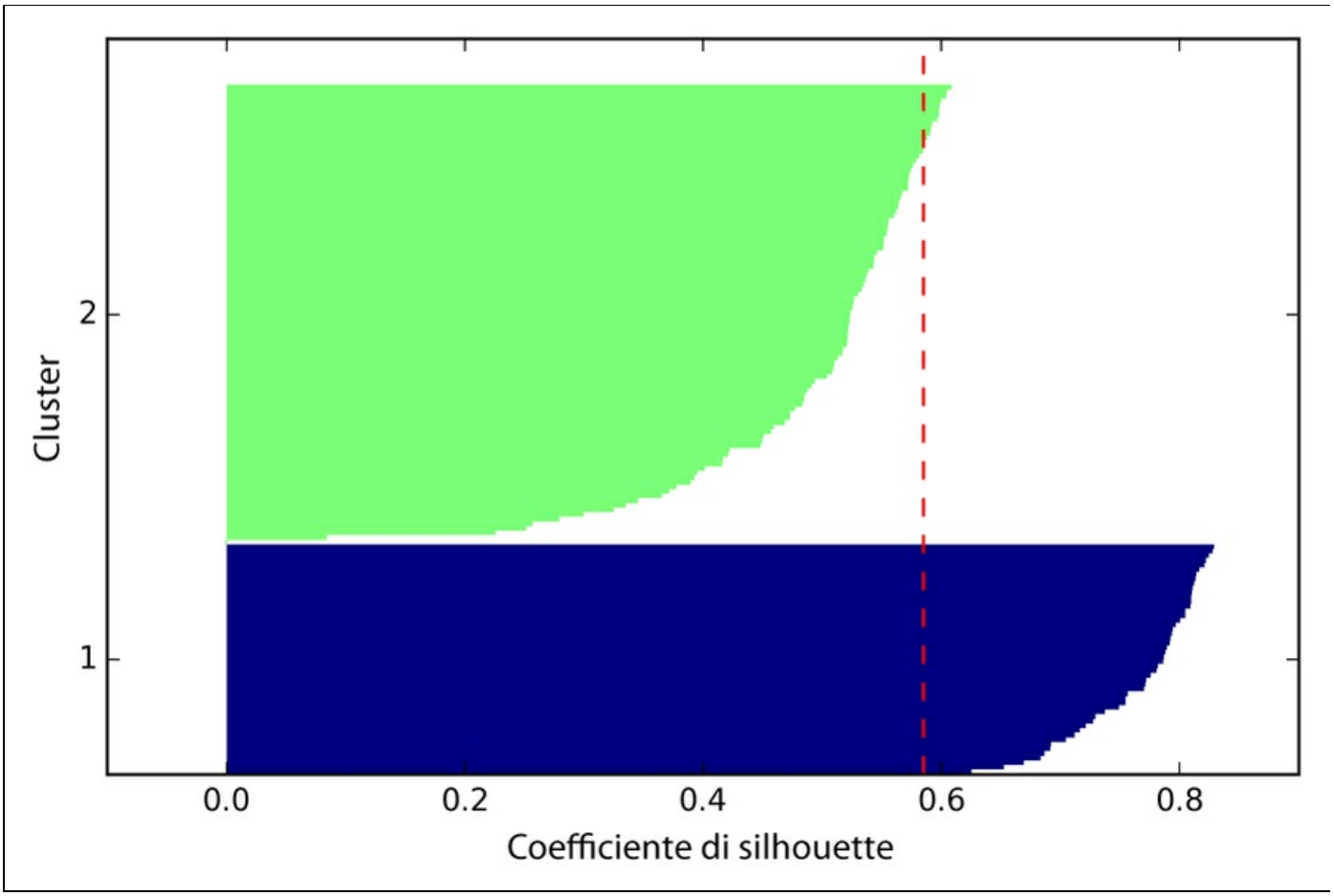


Figura 11.6

Organizzazione dei cluster come un albero gerarchico

In questo paragrafo esamineremo un approccio alternativo al clustering basato su un prototipo: il *clustering gerarchico*. Un vantaggio degli algoritmi a clustering gerarchico è il fatto che consentono di tracciare dei *dendrogrammi* (visualizzazione di un clustering gerarchico) che possono aiutare nell'interpretazione dei risultati, creando tassonomie significative. Un altro vantaggio di questo approccio gerarchico è il fatto che non abbiamo bisogno di specificare fin da subito il numero di cluster.

I due approcci principali al clustering gerarchico sono quello *agglomerativo* e quello *divisivo*. Nel clustering gerarchico divisivo partiamo con un cluster che comprende tutti i nostri campioni e lo suddividiamo iterativamente in cluster più piccoli, finché ciascun cluster non contiene un solo campione. In questo paragrafo, ci concentriamo sul clustering agglomerativo, che adotta l'approccio opposto. Partiamo con ciascun campione, che viene considerato come un singolo cluster, e uniamo i cluster a coppie per prossimità, fino a ottenere un unico cluster.

I due algoritmi standard per il clustering gerarchico agglomerativo sono quelli a collegamento singolo (*single linkage*) e a collegamento doppio (*complete linkage*). Utilizzando il collegamento semplice, calcoliamo le distanze fra i membri più simili di ciascuna coppia di cluster e uniamo i due cluster per i quali la distanza fra i membri più simili è la più piccola. L'approccio a collegamento completo è simile ma, invece di confrontare i membri più simili di ciascuna coppia di cluster, confrontiamo i membri più dissimili per eseguire la fusione. L'operazione è riassunta nella Figura 11.7.

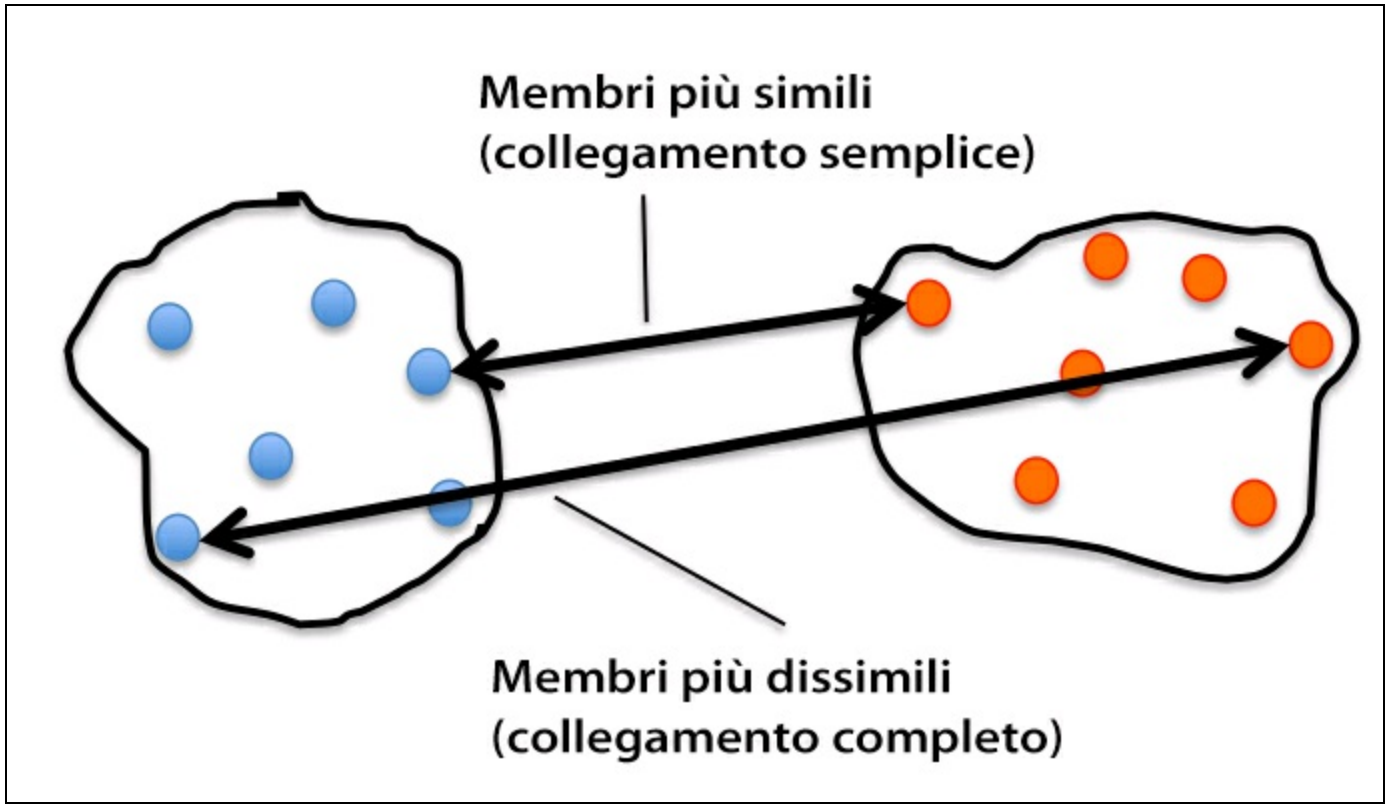


Figura 11.7

NOTA

Altri algoritmi comunemente utilizzati per il clustering gerarchico agglomerativo sono *average linkage* e *Ward's linkage*. Nel primo caso uniamo le coppie di cluster sulla base della minima distanza media fra tutti i membri del gruppo dei due cluster. Nel metodo Ward vengono fusi quei due cluster che conducono al minimo incremento del valore SSE interno del cluster.

In questo paragrafo, ci concentreremo sui clustering agglomerativi, utilizzando l'approccio a collegamento completo. Si tratta di una procedura iterativa che può essere riepilogata dai seguenti passi.

1. Calcolare la matrice delle distanze di tutti i campioni.
2. Rappresentare ciascun punto dei dati come un cluster singleton.
3. Unire i due cluster più vicini sulla base della distanza dei membri più dissimili (distanti).
4. Aggiornare la matrice della similarità.
5. Ripetere i Passi da 2 a 4 fino a ottenere un unico cluster.

Ora vedremo come calcolare la matrice delle distanze (Passo 1). Ma prima generiamo un campione di dati casuale con cui lavorare. Le righe rappresentano osservazioni differenti (ID da 0 a 4) e le colonne caratteristiche differenti (X, Y, Z) di questi campioni:

```
>>> import pandas as pd
>>> import numpy as np
```

```
>>> np.random.seed(123)
>>> variables = ['X', 'Y', 'Z']
>>> labels = ['ID_0','ID_1','ID_2','ID_3','ID_4']
>>> X = np.random.random_sample([5,3])*10
>>> df = pd.DataFrame(X, columns=variables, index=labels)
>>> df
```

Dopo aver eseguito il codice precedente, dovremmo avere la matrice delle distanze rappresentata nella Tabella 11.1.

Tabella 11.1

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

Esecuzione di un clustering gerarchico su una matrice delle distanze

Per calcolare la matrice delle distanze che funge da input dell'algoritmo di clustering gerarchico, utilizziamo la funzione `pdist` del modulo `spatial.distance` di SciPy:

```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...         pdist(df, metric='euclidean')),
...         columns=labels, index=labels)
>>> row_dist
```

Utilizzando il codice precedente, abbiamo calcolato la distanza euclidea fra ciascuna coppia di punti campione del nostro dataset sulla base delle caratteristiche X, Y e Z. Abbiamo fornito quale input alla funzione `squareform` la matrice delle distanze condensata (restituita da `pdist`) per creare una matrice simmetrica delle distanze a coppie, come illustrato nella Tabella 11.2.

Tabella 11.2

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

Poi applichiamo ai nostri cluster l'agglomerazione a collegamento completo, utilizzando la funzione `linkage` del modulo `cluster.hierarchy` di SciPy, che restituisce una cosiddetta *matrice dei collegamenti*.

Tuttavia, prima di richiamare la funzione linkage, è il caso di dare un'occhiata alla documentazione della funzione:

```
>>> from scipy.cluster.hierarchy import linkage
>>> help(linkage)
[…]
Parameters:
  y : ndarray
    A condensed or redundant distance matrix. A condensed
    distance matrix is a flat array containing the upper
    triangular of the distance matrix. This is the form
    that pdist returns. Alternatively, a collection of m
    observation vectors in n dimensions may be passed as
    an m by n array.
  method : str, optional
    The linkage algorithm to use. See the Linkage Methods
    section below for full descriptions.
  metric : str, optional
    The distance metric to use. See the distance.pdist
    function for a list of valid distance metrics.
 Returns:
  Z : ndarray
    The hierarchical clustering encoded as a linkage matrix.
[…]
```

Sulla base della descrizione della funzione, concludiamo che possiamo utilizzare come attributo di input una matrice delle distanze condensata (triangolare superiore) dalla funzione pdist. Alternativamente, potremmo anche fornire l'array dei dati iniziale e utilizzare la metrica euclidean come argomento della funzione in linkage. Tuttavia, non dovremmo utilizzare la matrice delle distanze squareform che abbiamo definito in precedenza, in quanto fornirebbe valori di distanza differenti da quelli previsti. Per riepilogare, ecco le tre situazioni possibili.

- *Approccio errato* – Con questo approccio, utilizziamo la matrice delle distanze a forma quadrata. Il codice è il seguente:

```
>>> from scipy.cluster.hierarchy import linkage
>>> row_clusters = linkage(row_dist,
...                        method='complete',
...                        metric='euclidean')
```

- *Approccio corretto* – Con questo approccio, utilizziamo la matrice delle distanze condensata, il codice è il seguente:

```
>>> row_clusters = linkage(pdist(df,
...                        metric='euclidean'),
...                        method='complete')
```

- *Approccio corretto* – Con questo approccio, utilizziamo la matrice dei campioni di input. Il codice è il seguente:

```
>>> row_clusters = linkage(df.values,
...                        method='complete',
...                        metric='euclidean')
```

Per dare un'occhiata più da vicino ai risultati del clustering, possiamo trasformarli in un pandas DataFrame (è meglio vederlo in IPython Notebook) nel

seguente modo:

```
>>> pd.DataFrame(row_clusters,
...     columns=['row label 1',
...              'row label 2',
...              'distance',
...              'no. of items in clust.'],
...     index=['cluster %d' %(i+1) for i in
...            range(row_clusters.shape[0])])
```

Come si può vedere nella Tabella 11.3, la matrice dei collegamenti è costituita da varie righe, ognuna delle quali rappresenta una fusione. La prima e la seconda colonna denotano i membri più dissimili di ciascun cluster e la terza riga rileva la distanza esistente fra questi membri. L'ultima colonna restituisce il conteggio dei membri di ciascun cluster.

Tabella 11.3

	Etichetta riga 1	Etichetta riga 2	Distanza	Numero oggetti nel cluster
Cluster 1	0	4	3.835396	2
Cluster 2	1	2	4.347073	2
Cluster 3	3	5	5.899885	3
Cluster 4	6	7	8.316594	5

Ora che abbiamo calcolato la matrice dei collegamenti, possiamo rappresentare i risultati sotto forma di un dendrogramma:

```
>>> from scipy.cluster.hierarchy import dendrogram
# make dendrogram black (part 1/2)
# from scipy.cluster.hierarchy import set_link_color_palette
# set_link_color_palette(['black'])
>>> row_dendr = dendrogram(row_clusters,
...                        labels=labels,
...                        # make dendrogram black (part 2/2)
...                        # color_threshold=np.inf
...                        )
>>> plt.tight_layout()
>>> plt.ylabel('Euclidean distance')
>>> plt.show()
```

Se state eseguendo il codice precedente, noterete che le ramificazioni del programma risultante sono rappresentate in colori differenti. Lo schema dei colori deriva da un elenco di colori di matplotlib che vengono ripresentati in modo ciclico per le soglie di distanza del dendrogramma. Per esempio, per rappresentare i dendrogrammi in nero, potete togliere il commento alle rispettive sezioni che sono presenti nel codice precedente.

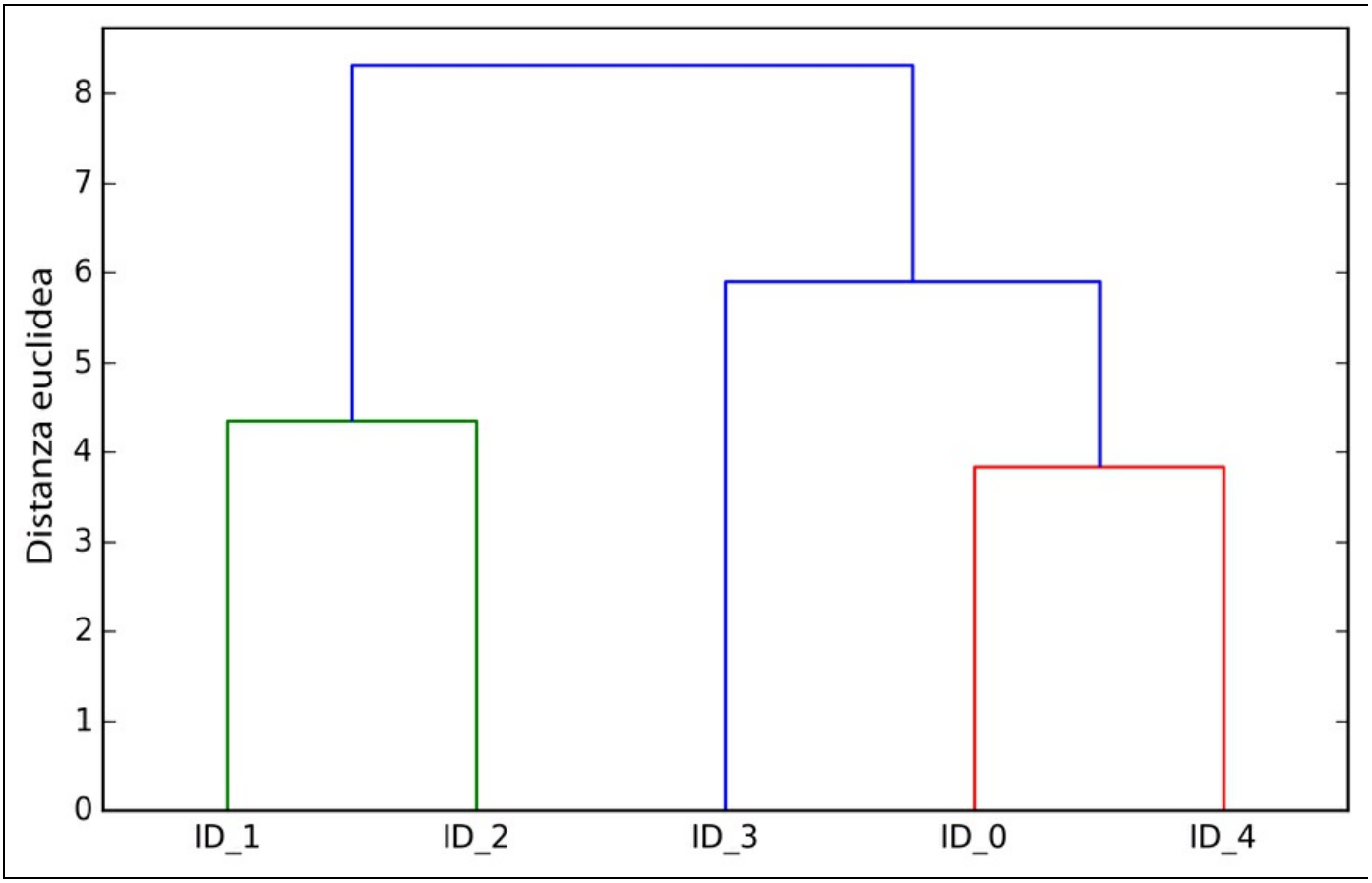


Figura 11.8

Questo dendrogramma riepiloga i diversi cluster che si sono formati durante il clustering gerarchico agglomerativo; per esempio, possiamo vedere che i campioni *ID_0* e *ID_4*, seguiti da *ID_1* e *ID_2*, sono quelli più simili sulla base della metrica della distanza euclidea.

Collegamento dei dendrogrammi a una mappa termica

Nelle applicazioni pratiche, i dendrogrammi di clustering gerarchico vengono frequentemente utilizzati in associazione a una *mappa termica*, che ci consente di rappresentare i singoli valori della matrice dei campioni con un codice di colori. In questo paragrafo, vedremo come collegare un dendrogramma a una mappa termica per ordinare opportunamente le righe di tale mappa.

Tuttavia, il collegamento di un dendrogramma a una mappa termica può essere piuttosto difficoltoso. Esaminiamo la procedura passo dopo passo.

1. Creiamo un nuovo oggetto `figure` e definiamo la posizione dell'asse *x*, dell'asse *y* e la larghezza e l'altezza del dendrogramma tramite l'attributo `add_axes`. Inoltre,

ruotiamo il dendrogramma di 90° in senso antiorario. Il codice è il seguente:

```
>>> fig = plt.figure(figsize=(8,8), facecolor='white')
>>> axd = fig.add_axes([0.09,0.1,0.2,0.6])
>>> row_dendr = dendrogram(row_clusters, orientation='right')
# note: for matplotlib >= v1.5.1, please use orientation='left'
```

2. Poi riordiniamo i dati nel DataFrame iniziale sulla base delle etichette di clustering, alle quali si può accedere dall'oggetto dendrogram, sostanzialmente un dizionario Python, tramite la chiave leaves. Il codice è il seguente:

```
>>> df_rowclust = df.ix[row_dendr['leaves'][::-1]]
```

3. Ora costruiamo la mappa termica dal DataFrame ordinato e la posizioniamo proprio in mezzo al dendrogramma:

```
>>> axm = fig.add_axes([0.23,0.1,0.6,0.6])
>>> cax = axm.matshow(df_rowclust,
...              interpolation='nearest', cmap='hot_r')
```

4. Infine, modifichiamo l'aspetto grafico della mappa termica eliminando i dettagli (ticks e spines). Inoltre, aggiungiamo una barra dei colori e assegnamo i nomi delle caratteristiche e dei campioni agli assi *x* e *y*. Il codice è il seguente:

```
>>> axd.set_xticks([])
>>> axd.set_yticks([])
>>> for i in axd.spines.values():
...     i.set_visible(False)
>>> fig.colorbar(cax)
>>> axm.set_xticklabels(['] + list(df_rowclust.columns))
>>> axm.set_yticklabels(['] + list(df_rowclust.index))
>>> plt.show()
```

Dopo aver seguito i passi precedenti, la mappa termica affiancherà il relativo dendrogramma (Figura 11.9).

Come possiamo vedere, l'ordine delle righe nella mappa termica riflette il clustering dei campioni del dendrogramma. Oltre a un dendrogramma semplice, la codifica a colori di ciascun campione e la funzionalità della mappa termica ci forniscono un buon riepilogo del dataset.

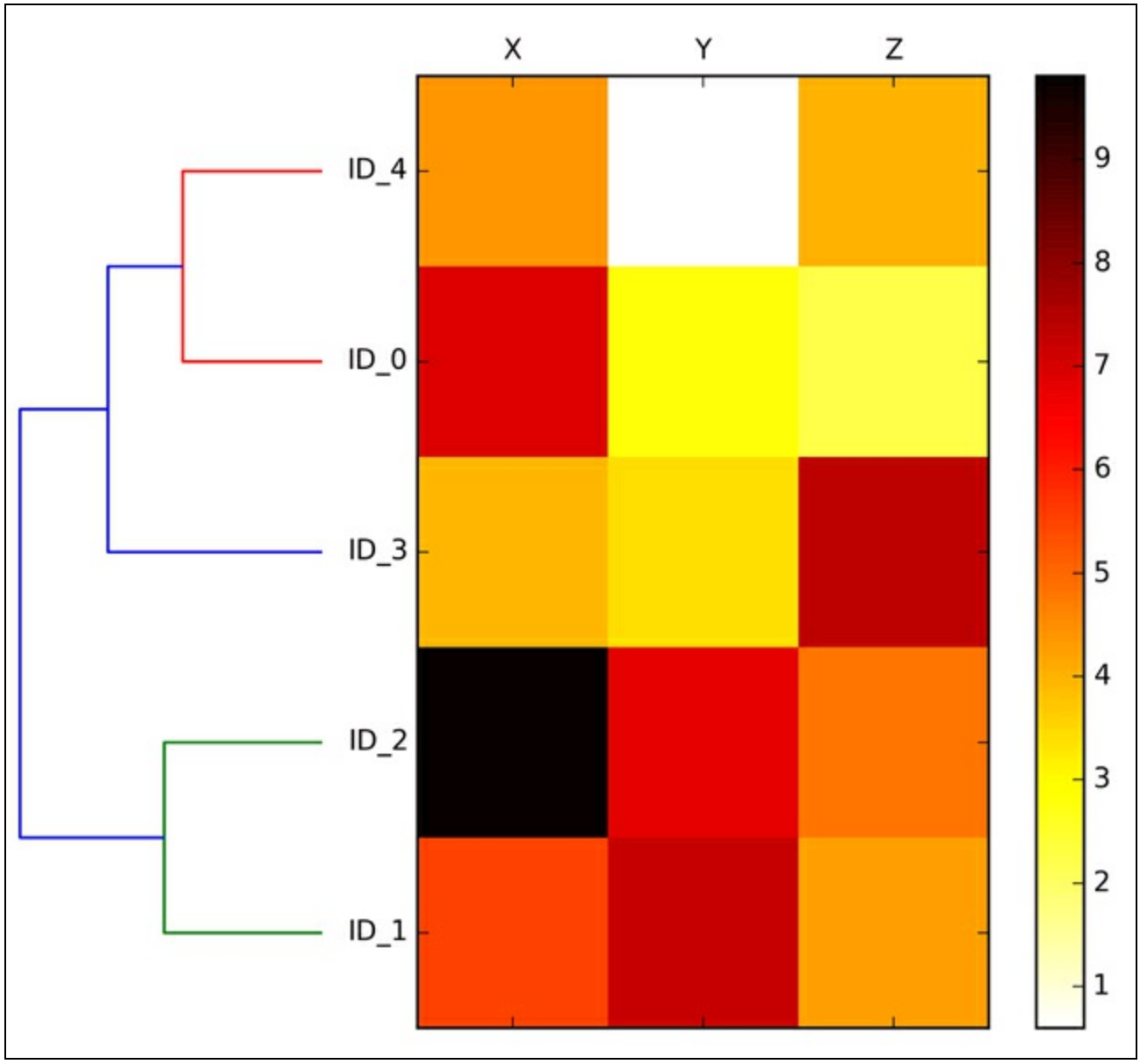


Figura 11. 9

Applicazione del clustering agglomerativo tramite scikit-learn

Abbiamo visto come eseguire un clustering gerarchico agglomerativo utilizzando SciPy. Tuttavia, esiste anche un'implementazione AgglomerativeClustering in scikit-learn, che ci consente di scegliere il numero di cluster che vogliamo ottenere. Questo è utile se vogliamo potare l'albero dei cluster gerarchici. Impostando il parametro n_cluster a 2, suddivideremo il cluster in due gruppi utilizzando lo stesso approccio a collegamento completo basato sulla metrica della distanza euclidea che abbiamo impiegato in precedenza:

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> ac = AgglomerativeClustering(n_clusters=2,
```

```
...                         affinity='euclidean',
...                         linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print('Cluster labels: %s' % labels)
Cluster labels: [0 1 1 0 0]
```

Osservando le etichette previste per le classi, possiamo vedere che il primo, il quarto e il quinto campione (*ID_0*, *ID_3* e *ID_4*) sono stati assegnati a un cluster (0) e che i campioni *ID_1* e *ID_2* sono stati assegnati a un secondo cluster (1), il che è coerente con i risultati che abbiamo osservato nel dendrogramma.

Individuazione delle regioni a elevata densità tramite DBSCAN

Anche se non è possibile trattare il gran numero di algoritmi di clustering disponibili, è il caso di introdurre almeno un altro approccio al clustering: *DBSCAN* (*Density-Based Spatial Clustering of Applications with Noise*). Il concetto di densità di DBSCAN si definisce come il numero di punti che ricadono entro un determinato raggio ε.

In DBSCAN, viene assegnata una determinata etichetta a ciascun campione (punto) utilizzando i seguenti criteri.

- Un punto è considerato *punto core* se almeno un determinato numero (*MinPts*) dei punti confinanti rientra nel raggio specificato ε.
- Un *punto di confine* è un punto che ha meno vicini rispetto a *MinPts* che sono entro ε, ma che entra comunque nel raggio ε di distanza da un punto core.
- Tutti gli altri punti che non sono né core, né di confine sono considerati *punti di rumore*.

Dopo aver etichettato i punti come core, di confine o di rumore, l'algoritmo DBSCAN può essere riepilogato in due semplici passi.

1. Formare un cluster distinto per ciascun punto core o gruppo connesso di punti core (i punti core sono connessi se non sono distanti fra loro più di ε).
2. Assegnare ciascun punto di confine al cluster del corrispondente punto core.

Per comprendere meglio i risultati di DBSCAN prima di vedere l'implementazione, riepiloghiamo tramite la Figura 11.10 che cosa abbiamo imparato sui punti core, i punti di confine e i punti di rumore.

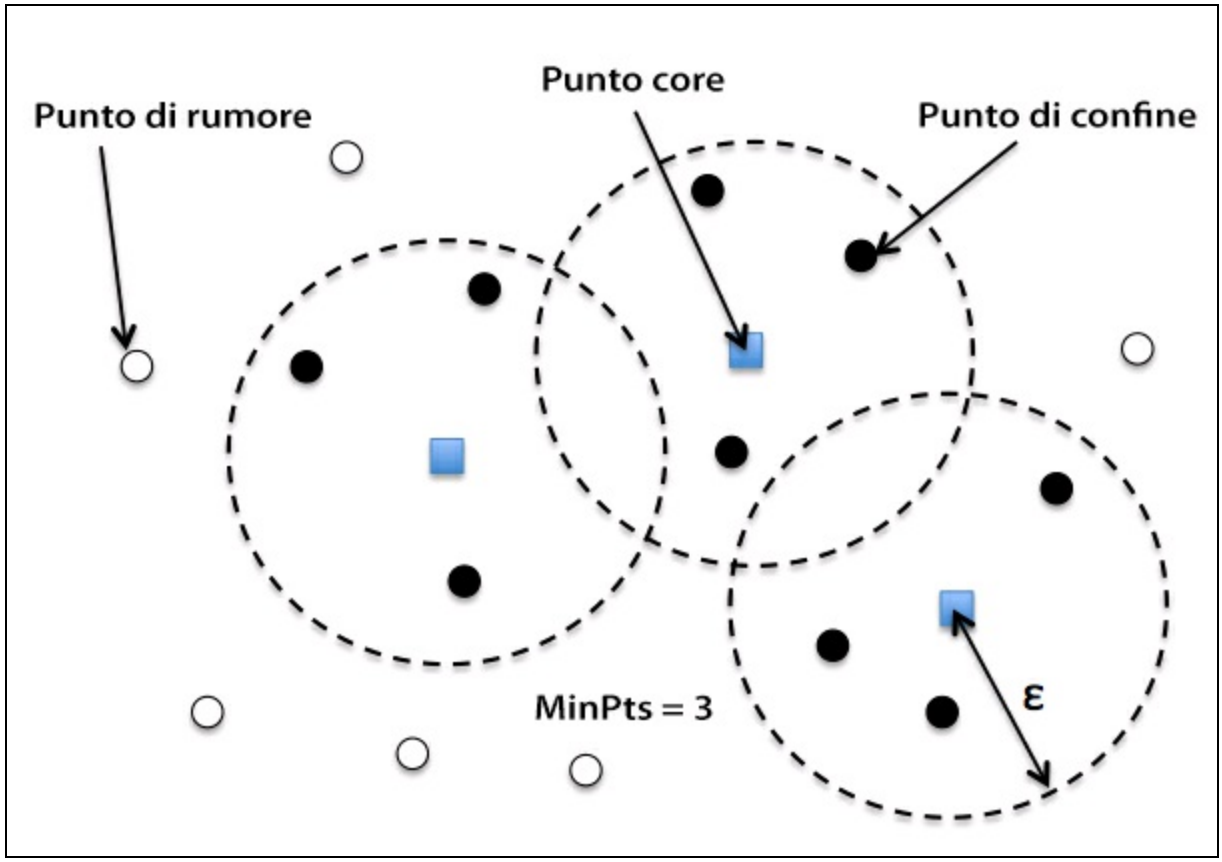


Figura 11.10

Uno dei principali vantaggi dell'uso di DBSCAN è il fatto che non presuppone che i cluster abbiano una forma sferica come in k-means. Inoltre, DBSCAN è differente dal clustering k-means ed è gerarchico, per il fatto che non assegna necessariamente ciascun punto a un cluster, ma è in grado di rimuovere i punti di rumore.

Per un esempio più illustrativo, creiamo un nuovo dataset di strutture a mezzaluna, per confrontare il funzionamento del clustering k-means, gerarchico e DBSCAN:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=200,
...                   noise=0.05,
...                   random_state=0)
>>> plt.scatter(X[:,0], X[:,1])
>>> plt.show()
```

Come possiamo vedere dalla Figura 11.11, esistono due gruppi di punti a mezzaluna, ognuno dei quali è costituito da 100 campioni.

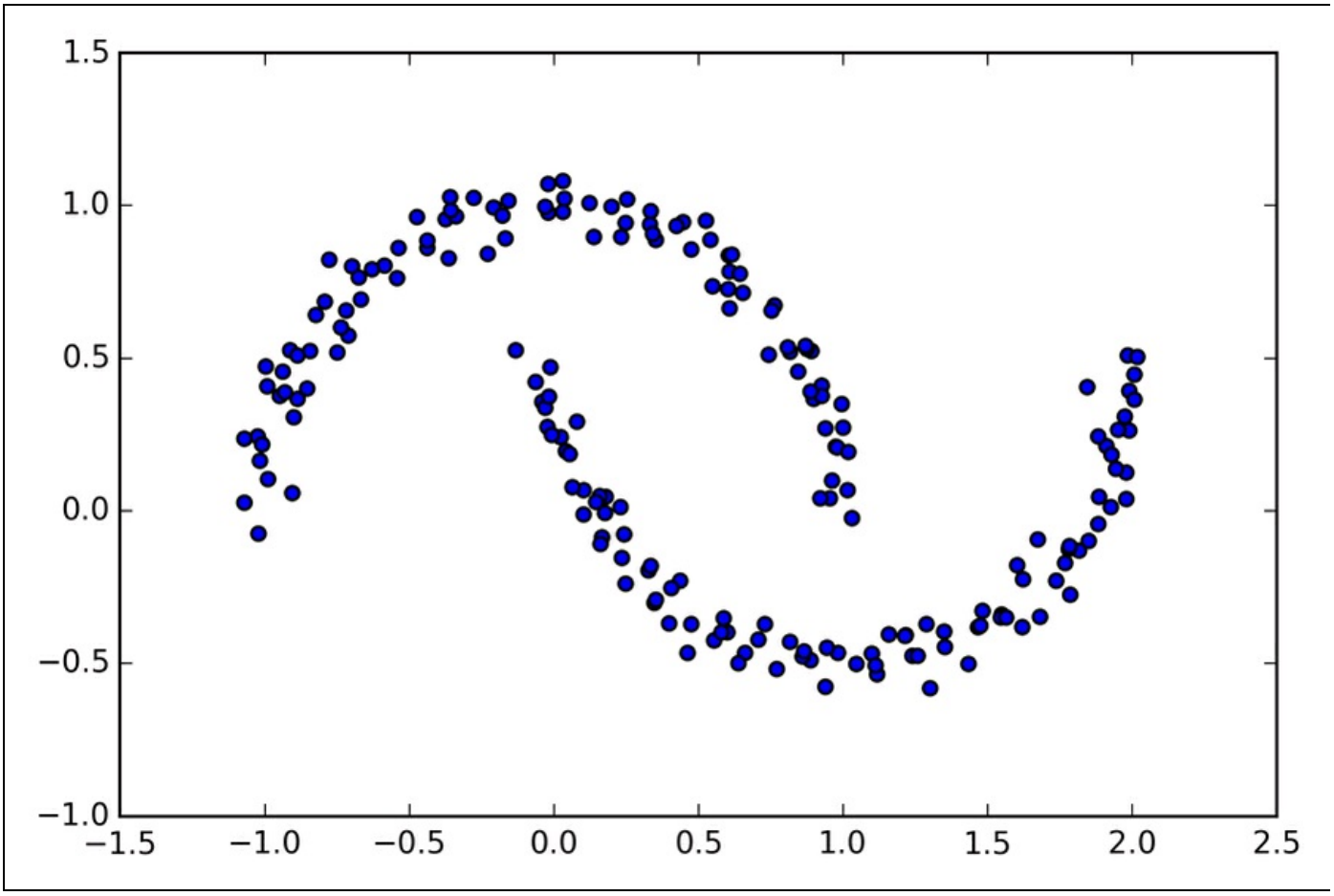


Figura 11.11

Inizieremo a utilizzare l'algoritmo k-means e il clustering a collegamento completo per vedere se uno degli algoritmi di clustering di cui abbiamo parlato è in grado di identificare con successo come cluster distinti le forme a mezzaluna. Il codice è il seguente:

```
>>> f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8,3))
>>> km = KMeans(n_clusters=2,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> ax1.scatter(X[y_km==0,0],
...             X[y_km==0,1],
...             c='lightblue',
...             marker='o',
...             s=40,
...             label='cluster 1')
>>> ax1.scatter(X[y_km==1,0],
...             X[y_km==1,1],
...             c='red',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> ax1.set_title('K-means clustering')
>>> ac = AgglomerativeClustering(n_clusters=2,
...                              affinity='euclidean',
...                              linkage='complete')
>>> y_ac = ac.fit_predict(X)
>>> ax2.scatter(X[y_ac==0,0],
...             X[y_ac==0,1],
...             c='lightblue',
...             marker='o',
...             s=40,
...             label='cluster 1')
```

```
>>> ax2.scatter(X[y_ac==1,0],
...             X[y_ac==1,1],
...             c='red',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> ax2.set_title('Agglomerative clustering')
>>> plt.legend()
>>> plt.show()
```

Sulla base dei risultati del clustering, possiamo vedere che l'algoritmo k-means non è in grado di separare i due cluster e che l'algoritmo di clustering gerarchico entra in crisi con forme così complesse (Figura 11.12).

Infine, proviamo ad applicare l'algoritmo DBSCAN su questo dataset per vedere se è in grado di individuare i due cluster a mezzaluna utilizzando un approccio basato sulla densità:

```
>>> from sklearn.cluster import DBSCAN
>>> db = DBSCAN(eps=0.2,
...             min_samples=5,
...             metric='euclidean')
>>> y_db = db.fit_predict(X)
>>> plt.scatter(X[y_db==0,0],
...             X[y_db==0,1],
...             c='lightblue',
...             marker='o',
...             s=40,
...             label='cluster 1')
>>> plt.scatter(X[y_db==1,0],
...             X[y_db==1,1],
...             c='red',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> plt.legend()
>>> plt.show()
```

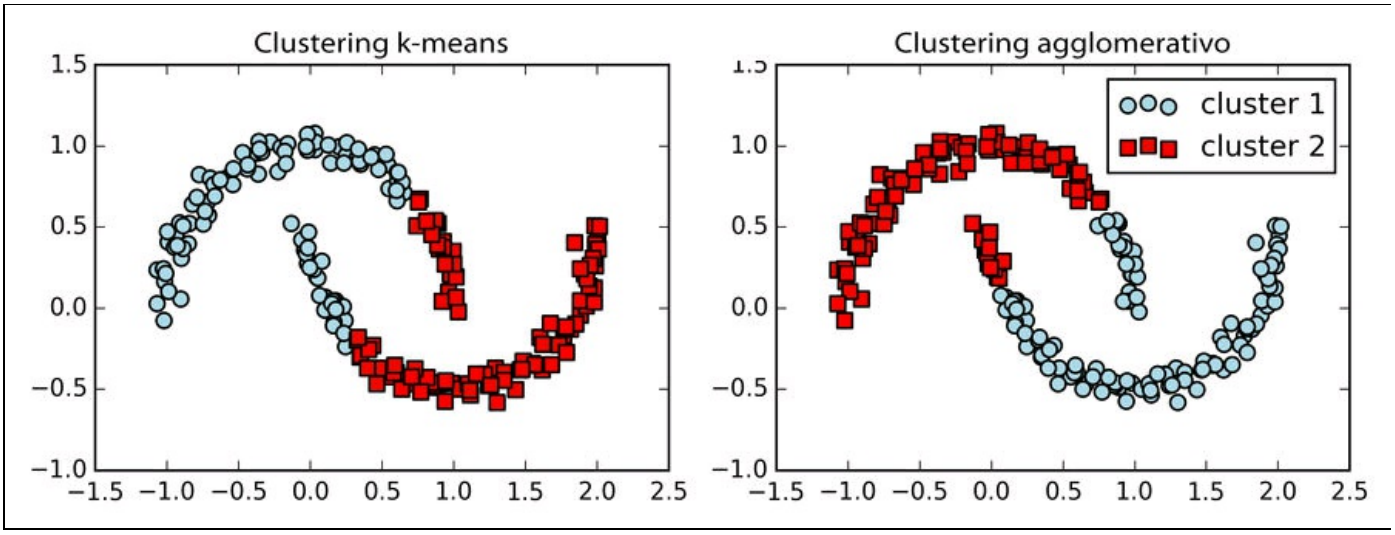


Figura 11.12

L'algoritmo DBSCAN è in grado di rilevare con successo le forme a mezzaluna (Figura 11.13), il che evidenzia uno dei suoi punti di forza, ovvero il clustering di dati di forma arbitraria.

Tuttavia, occorre anche notare che DBSCAN presenta degli svantaggi. Con un numero crescente di caratteristiche nel dataset, dato un set di addestramento di dimensioni fisse, si incrementa l'effetto negativo della *maledizione della dimensionalità*. Questo problema è particolarmente evidente se utilizziamo la metrica della distanza euclidea. Tuttavia, il problema della *maledizione della dimensionalità* non è specifico di DBSCAN; può pregiudicare il funzionamento anche di altri algoritmi di clustering che utilizzano la metrica della distanza euclidea, per esempio gli algoritmi k-means e di clustering gerarchico. Inoltre, DBSCAN prevede due iperparametri (MinPts e $\mathcal{E}$) che devono essere ottimizzati per ottenere risultati di clustering corretti. La ricerca di una buona combinazione di MinPts e $\mathcal{E}$ può essere problematica se le differenze di densità nel dataset non sono particolarmente evidenti.

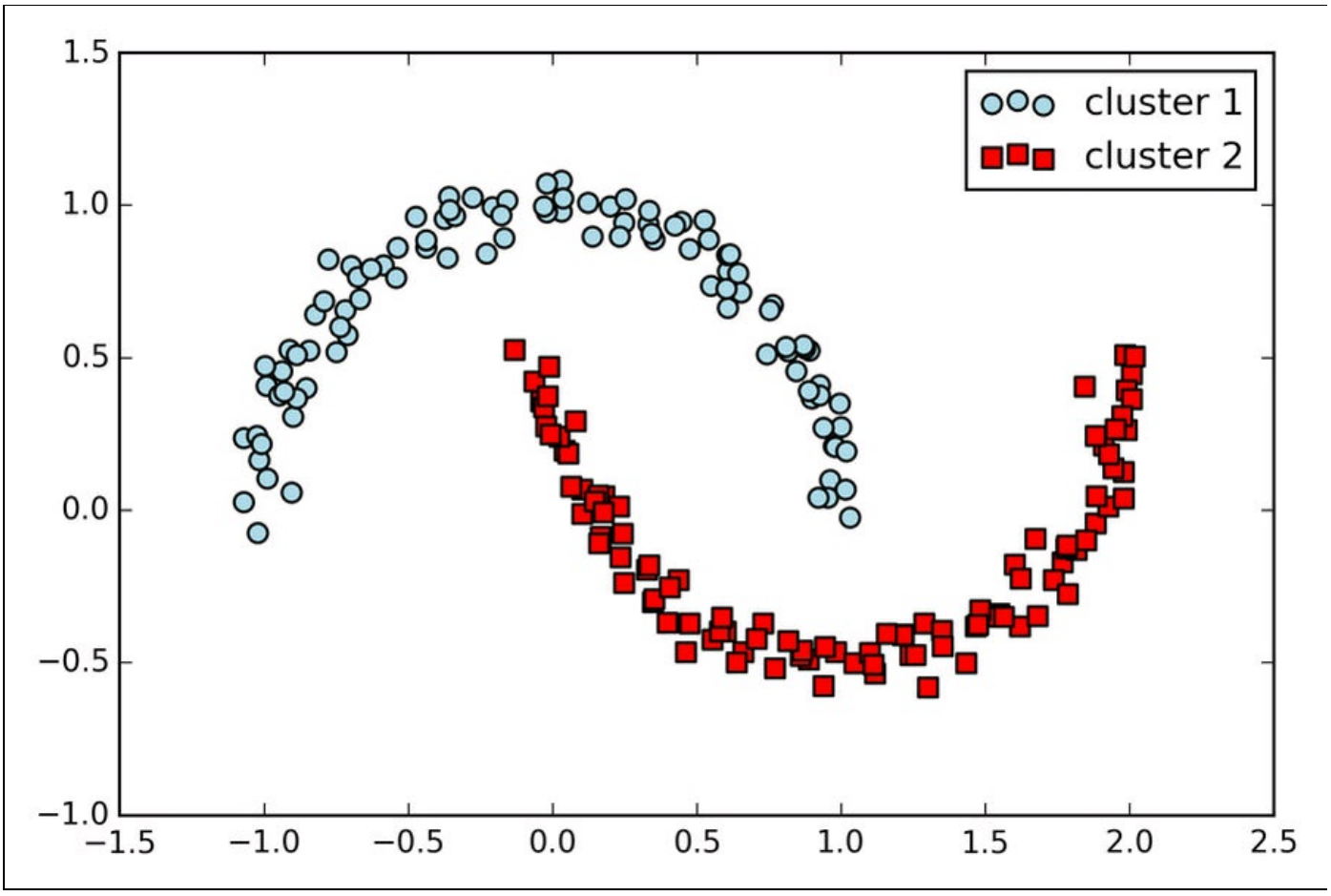


Figura 11.13

NOTA

Finora abbiamo visto tre delle categorie fondamentali di algoritmi di clustering: clustering basato su prototipi con k-means, clustering gerarchico agglomerativo e clustering a densità con DBSCAN. Tuttavia, vale la pena di menzionare anche una quarta classe di algoritmi di clustering più avanzati, che non abbiamo trattato in questo capitolo: il *clustering basato su grafici*. Probabilmente i membri preminenti della famiglia del clustering basato su grafici sono gli

algoritmi di clustering spettrale. Sebbene esistano varie implementazioni del clustering spettrale, tutte hanno in comune l'uso degli autovettori di una matrice di similarità per derivare le relazioni nei cluster. Poiché il clustering spettrale non rientra negli scopi di questo libro, rimandiamo all'eccellente guida di Ulrike von Luxburg (U. Von Luxburg, *A Tutorial on Spectral Clustering, in "*Statistics and computing", 17(4):395–416, 2007). Tale pubblicazione è disponibile gratuitamente presso arXiv l'indirizzo http://arxiv.org/pdf/0711.0189v1.pdf.

Notate che, nella pratica, non sempre è facile capire quale algoritmo si comporta meglio degli altri su un determinato dataset, specialmente quando i dati sono dotati di più dimensioni che ne rendono difficile o impossibile la visualizzazione. Inoltre, è importante enfatizzare il fatto che un clustering di successo non dipende solo dall'algoritmo e dai suoi iperparametri. Piuttosto saranno la scelta di una metrica della distanza appropriata e l'uso della conoscenza del dominio ad aiutarci a sperimentare le varie configurazioni.

Riepilogo

In questo capitolo abbiamo parlato di tre diversi algoritmi di clustering che possono aiutarci a individuare le strutture nascoste o le informazioni contenute nei dati. Siamo partiti da un approccio basato sul prototipo, k-means, che suddivide in cluster i campioni tramite forme sferiche sulla base del numero specificato di centroidi del cluster. Poiché il clustering è un metodo senza supervisione, non abbiamo la fortuna di conoscere le etichette vere per valutare le prestazioni di un modello. Pertanto, dobbiamo contare sull'uso di metriche prestazionali intrinseche, come il metodo Elbow o l'analisi della silhouette, nel tentativo di quantificare la qualità del clustering.

Poi abbiamo osservato un approccio differente al clustering: il clustering gerarchico agglomerativo. Il clustering gerarchico non richiede di specificare il numero di cluster prima di iniziare e il risultato può essere visualizzato tramite una rappresentazione a dendrogramma, che può aiutare con l'interpretazione dei risultati. L'ultimo algoritmo di clustering che abbiamo visto in questo capitolo è DBSCAN, in grado di raggruppare i punti sulla base delle densità locali e di gestire i valori anomali e di identificare le forme non globulari.

Dopo questa escursione nel campo dell'apprendimento senza supervisione, è giunto il momento di introdurre alcuni dei più interessanti algoritmi di machine learning per l'apprendimento con supervisione: le reti neurali artificiali multilivello. Dopo la loro recente "resurrezione", le reti neurali sono tornate a essere un argomento avvincente nel campo della ricerca in ambito del machine learning. Grazie ad algoritmi recentemente sviluppati di apprendimento profondo, le reti neurali sono attualmente lo stato dell'arte per molti compiti complessi, come la classificazione delle immagini e il riconoscimento del parlato. Nel Capitolo 12, *Reti neurali artificiali per il riconoscimento delle immagini*, costruiremo partendo da zero una nostra rete neurale multilivello. Nel Capitolo 13, *Parallelizzare l'addestramento delle reti neurali con Theano*, introdurremo alcune potenti librerie che possono aiutarci ad addestrare in modo più efficiente le complesse architetture di rete.

Reti neurali artificiali per il riconoscimento delle immagini

Il *deep learning* è un argomento sempre più dibattuto ed è, senza alcun dubbio, l'argomento più affascinante nel campo del machine learning. Il deep learning può essere considerato come un insieme di algoritmi che sono stati sviluppati per addestrare con particolare efficienza reti neurali artificiali composte da più livelli. In questo capitolo, descriveremo i concetti di base delle reti neurali artificiali, in modo da fornire tutto il background necessario per esplorare ulteriormente i campi più interessanti della ricerca nell'ambito del machine learning, esaminando anche le librerie di learning di Python più avanzate che sono attualmente ancora in fase di sviluppo.

Tratteremo i seguenti argomenti.

- Descrizione concettuale delle reti neurali multilivello.
- Addestramento delle reti neurali per la classificazione delle immagini.
- Implementazione del potente algoritmo di retropropagazione.
- Debugging delle implementazioni delle reti neurali.
- Modellazione di funzioni complesse con reti neurali artificiali

Modellare funzioni complesse con reti neurali artificiali

Nei primi capitoli di questo libro, abbiamo iniziato il nostro viaggio nell'ambito degli algoritmi di machine learning con i neuroni artificiali (Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*). I neuroni artificiali rappresentano gli elementi costitutivi delle reti neurali artificiali multilivello, di cui parleremo in questo capitolo. Il concetto su cui si basano le reti neurali artificiali deriva da ipotesi e modelli del funzionamento del cervello umano, quando si trova a dover risolvere problemi complessi. Sebbene le reti neurali artificiali abbiano acquisito una grande popolarità nel corso degli ultimi anni, i primi studi in questo senso risalgono agli anni Quaranta, quando Warren McCulloch e Walter Pitt descrissero il funzionamento dei neuroni. Tuttavia, nei decenni che seguirono la prima implementazione del modello del *neurone di McCulloch-Pitt*, ovvero il perceptron di Rosenblatt, negli anni Cinquanta, molti ricercatori ed esperti di machine learning iniziarono lentamente a perdere interesse nelle reti neurali, in quanto nessuno aveva una buona soluzione per l'addestramento di una rete neurale multilivello. Alla fine, l'interesse nei confronti delle reti neurali si è ravvivato nel 1986, quando D.E. Rumelhart, G.E. Hinton e R.J. Williams furono coinvolti nella riscoperta e diffusione dell'algoritmo di retropropagazione (*backpropagation*) per addestrare in modo più efficiente le reti neurali, argomento di cui parleremo più in dettaglio in questo capitolo (David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams (1986), *Learning Representations by Back-propagating Errors*, in "Nature" 323 (6088): 533–536).

Nel corso del decennio precedente, vi sono state molte grandi scoperte nel campo degli algoritmi che oggi chiamiamo di *feature detector*, che possono essere utilizzati per creare rilevatori di caratteristiche a partire da dati senza etichetta, in modo da pre-addestrare le reti neurali profonde, ovvero costituite da più livelli. Le reti neurali sono un argomento interessante non solo in ambito accademico, ma anche per grandi utilizzatori di tecnologie come Facebook, Microsoft e Google, i quali investono pesantemente nelle reti neurali artificiali e nella ricerca nell'ambito del deep learning. Al momento attuale, le reti neurali complesse, alimentate dagli algoritmi di deep learning, sono considerate lo stato dell'arte nella soluzione dei problemi complessi come il riconoscimento delle immagini e del parlato. Fra gli

esempi più noti di prodotti della vita quotidiana che funzionano grazie al deep learning vi sono la ricerca per immagini di Google e anche Google Translate, un'applicazione per smartphone in grado di riconoscere automaticamente il testo nelle immagini ed eseguire una traduzione in tempo reale in venti diverse lingue (http://googleresearch.blogspot.com/2015/07/how-google-translate-squeezes-deep.html).

Ma molte altre e interessanti applicazioni delle reti neurali profonde sono attualmente in fase di sviluppo presso le principali società che si occupano di alte tecnologie, per esempio DeepFace di Facebook per il tagging delle immagini (Y. Taigman, M. Yang, M. Ranzato e L. Wolf DeepFace, *Closing the gap to human-level performance in face verification*, in "Computer Vision and Pattern Recognition CVPR", 2014 IEEE Conference, pp. 1701–1708) e DeepSpeech di Baidu, che è in grado di gestire le richieste vocali in lingua mandarina (A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. *DeepSpeech: Scaling up end-to-end speech recognition*, arXiv preprint arXiv:1412.5567, 2014). Inoltre, l'industria farmaceutica ha recentemente iniziato a utilizzare tecniche di deep learning per la scelta dei principi attivi e per la previsione della tossicità e la ricerca ha dimostrato che queste nuove tecniche superano notevolmente le prestazioni dei metodi tradizionali di screening virtuale (T. Unterthiner, A. Mayr, G. Klambauer e S. Hochreiter, *Toxicity prediction using deep learning*, arXiv preprint arXiv:1503.01445, 2015).

Ripasso sulle reti neurali monolivello

Questo capitolo tratta interamente l'argomento delle reti neurali multilivello: come funzionano e come addestrarle per risolvere problemi complessi. Tuttavia, prima di approfondire una specifica architettura di rete neurale multilivello, è il caso di ripassare alcuni dei concetti riguardanti reti neurali monolivello di cui abbiamo parlato nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, e in particolare l'algoritmo *Adaline* (*ADAptive LInear NEuron*) che è illustrato nella Figura 12.1.

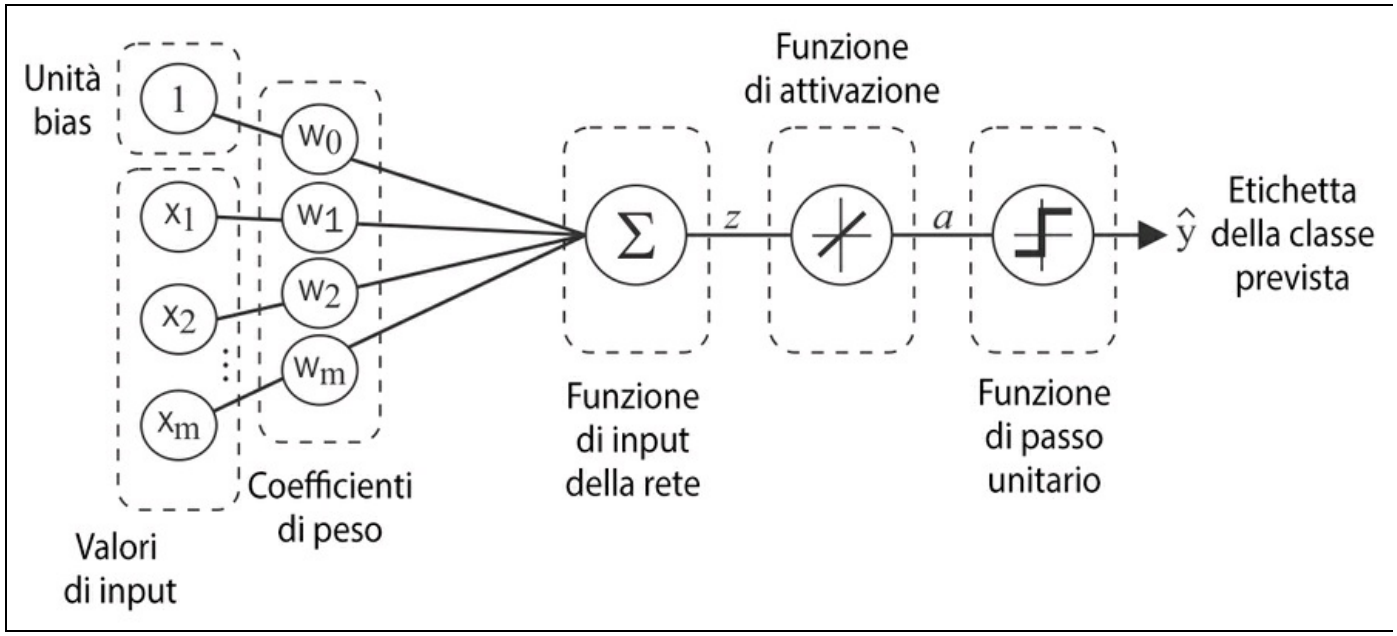


Figura 12.1

Nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, abbiamo implementato l'algoritmo Adaline per svolgere una classificazione binaria e abbiamo utilizzato un algoritmo di ottimizzazione a *discesa del gradiente* per imparare i coefficienti di peso del modello. A ogni epoch (passo di addestramento) abbiamo aggiornato il vettore dei pesi $\boldsymbol{w}$ utilizzando la seguente regola:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}, \text{ where } \Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w})$$

In altre parole, abbiamo calcolato il gradiente sulla base dell'intero insieme di addestramento e abbiamo aggiornato i pesi del modello compiendo un passo nella direzione opposta del gradiente $\nabla J(\mathbf{w})$. Per trovare i pesi ottimali del modello, abbiamo ottimizzato una funzione obiettivo che abbiamo definito come la funzione di costo *SSE* (*Sum of Squared Errors*) $J(\mathbf{w})$. Inoltre, abbiamo moltiplicato il gradiente per un fattore, il tasso di apprendimento η, che abbiamo scelto con cura per equilibrare la velocità di apprendimento e il rischio di fallire nell'individuazione del minimo globale della funzione di costo.

Nell'ottimizzazione della discesa del gradiente, abbiamo aggiornato tutti i pesi simultaneamente dopo ogni epoch e abbiamo definito la derivata parziale di ciascun peso w_j nel vettore dei pesi $\boldsymbol{w}$ nel seguente modo:

$$\frac{\partial}{\partial w_j} J(\boldsymbol{w}) = \sum_i \left(y^{(i)} - a^{(i)} \right) x_j^{(i)}$$

Qui $y^{(i)}$ è l'etichetta della classe di destinazione di un determinato campione $x^{(i)}$ e $a^{(i)}$ è l'attivazione del neurone, che, nel caso specifico di Adaline, è una funzione lineare. Inoltre, abbiamo definito la *funzione di attivazione* $\phi(\cdot)$ nel seguente modo:

$$\phi(z) = z = a$$

Qui l'input z è una combinazione lineare dei pesi che connettono il livello di input con quello di output:

$$z = \sum_j w_j x_j = \boldsymbol{w}^T \boldsymbol{x}$$

Anche se abbiamo utilizzato l'attivazione $\phi(z)$ per calcolare l'aggiornamento del gradiente, abbiamo implementato anche una *funzione soglia* (funzione Heaviside) per comprimere l'output da valori continui in etichette delle classi binarie per consentire la previsione:

$$\hat{y} = \begin{cases} 1 & if\ g(z) \geq 0 \\ -1 & altrimenti \end{cases}$$

NOTA

Notate che sebbene Adaline sia costituito da due livelli, un livello di input e un livello di output, si parla di rete monolivello, poiché fra questi due livelli esiste un solo collegamento.

Introduzione all'architettura di una rete neurale multilivello

In questo paragrafo vedremo come connettere più neuroni per formare una *rete neurale multilivello ad avanzamento* (multi-layer feedforward neural network); questo particolare tipo di rete è chiamata anche *perceptron multilivello* (MPL – multi-layer perceptron). La Figura 12.2 spiega il concetto di un MLP a tre livelli: un livello di input, un livello nascosto e un livello di output. Le unità presenti nel livello nascosto sono interamente connesse al livello di input e il livello di output è interamente connesso al livello nascosto. Se tale rete ha più di un livello nascosto, la chiamiamo rete neurale artificiale *profonda* (*deep artificial neural network*).

NOTA

Potremmo aggiungere un numero arbitrario di livelli nascosti al MLP per creare architetture di rete ancora più profonde. Ma nella pratica, possiamo considerare il numero di livelli e unità di

una rete neurale come se si trattasse di iperparametri aggiuntivi che vogliamo ottimizzare per un determinato compito, utilizzando la convalida incrociata di cui abbiamo parlato nel Capitolo 6, *Valutazione dei modelli e ottimizzazione degli iperparametri*. Tuttavia, i gradienti degli errori che calcoleremo successivamente tramite la retropropagazione diventeranno sempre più piccoli a mano a mano che alla rete vengono aggiunti ulteriori livelli. Questo problema del gradiente invisibile rende più complesso l'apprendimento del modello. Pertanto, sono stati sviluppati speciali algoritmi per pre-addestrare tali strutture di rete neurali profonde con quello che va sotto il nome di *deep learning*.

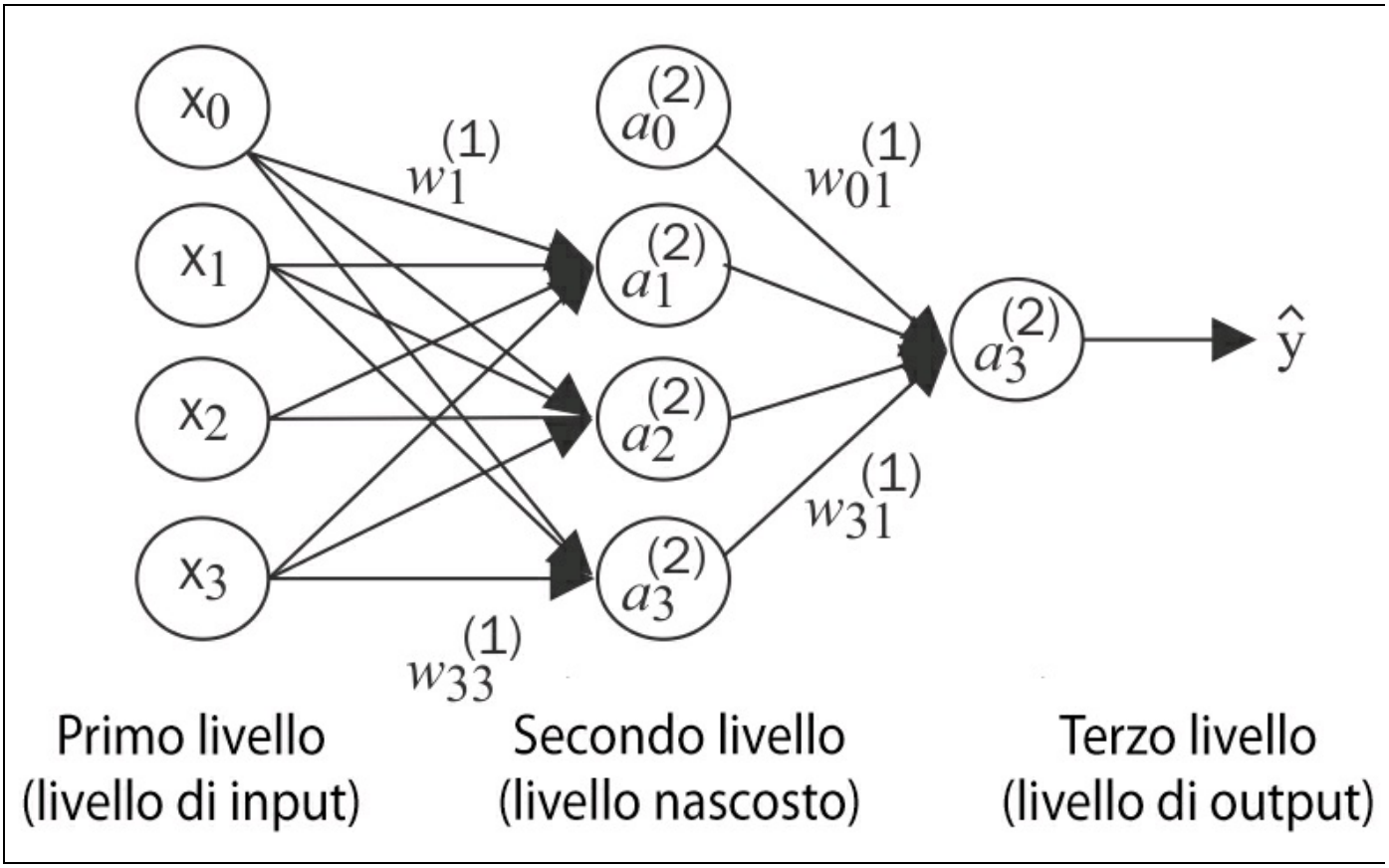


Figura 12.2

Come si può vedere dalla Figura 12.2, denotiamo l'unità d'attivazione *i*-esima nel livello *l*-esimo come $a_i^{(l)}$ e le unità di attivazione $a_0^{(1)}$ e $a_0^{(2)}$ sono unità di bias, impostate a 1. L'attivazione delle unità nel livello di input è costituita solo dal suo input più l'unità di bias:

$$a^{(1)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}$$

Ogni unità del livello *l* viene connessa a tutte le unità del livello *l*+1 tramite un coefficiente di peso. Per esempio, la connessione fra l'unità *k*-esima nel livello *l* con l'unità *j*-esima nel livello *l*+1 può essere scritta come $w_{j,k}^{(l)}$. Notate che l'indice *i* di $x_m^{(i)}$ identifica il campione *i*-esimo e non il livello *i*-esimo. Nei prossimi paragrafi, per non complicare troppo le cose, ometteremo l'indice *i*.

Anche se un'unità nel livello di output sarebbe sufficiente per un compito di classificazione binaria, vediamo una forma più generale di una rete neurale nella Figura 12.2 precedente, che ci consente di svolgere una classificazione multiclasse tramite una generalizzazione della tecnica *OvA* (*One-vs-All*). Per comprendere meglio il suo funzionamento, rimandiamo alla rappresentazione *one-hot* delle variabili categoriche che abbiamo introdotto nel Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione*. Per esempio, codificheremo le tre etichette delle classi dell'ormai noto dataset del fiore Iris (0=Setosa, 1=Versicolor, 2=Virginica) nel seguente modo:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Questa rappresentazione a vettore one-hot ci consente di occuparci dei compiti di classificazione con un numero arbitrario di etichette delle classi univoche presenti nel set di addestramento.

Chi non fosse avvezzo alle rappresentazioni di reti neurali, potrebbe trovare piuttosto ostica la terminologia relativa agli indici (fra tutti gli apici e i pedici). Qualcuno potrebbe chiedersi perché avremmo scritto $a^{(i)}$ e non $w_{k,j}^{(l)}$ per far riferimento al coefficiente di peso che connette la *k*-esima unità del livello *l* con la

j-esima unità del livello *l*+1. Ciò che può sembrare piuttosto confuso a prima vista acquisirà maggior senso nei prossimi paragrafi, quando trasformeremo in vettore la rappresentazione della rete neurale. Per esempio, riassumeremo i pesi che connettono il livello di input e quello nascosto con una matrice $W^{(1)} \in \mathbb{R}^{h \times [m+1]}$, dove *h* è il numero di unità nascoste e *m*+1 è il numero di unità di input più l'unità di bias. Poiché è importante comprendere questo concetto per poter seguire il contenuto di questo capitolo, riepiloghiamo ciò che abbiamo già trattato in un'illustrazione descrittiva di un perceptron multilivello 3-4-3 semplificato (Figura 12.3).

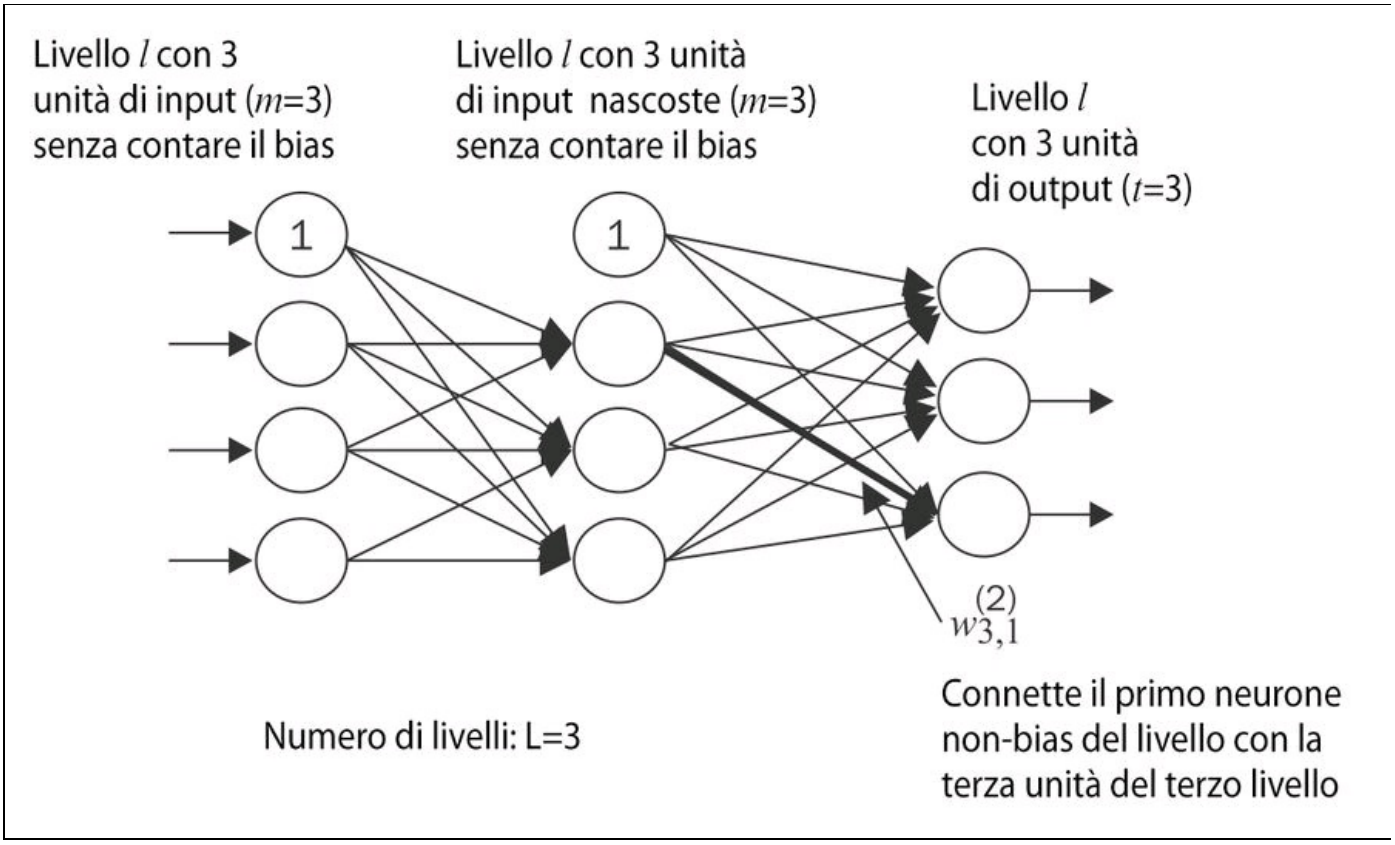


Figura 12.3

Attivazione di una rete neurale tramite propagazione in avanti

In questo paragrafo descriveremo il processo di *propagazione in avanti* per calcolare l'output di un modello MLP. Per comprendere come rientra nel contesto dell'apprendimento di un modello MLP, riepiloghiamo la procedura di apprendimento MLP in tre semplici passi.

1. Partendo dal livello di input, propaghiamo in avanti gli schemi dei dati di apprendimento tramite la rete, in modo da generare un output.
2. Sulla base dell'output della rete, calcoliamo l'errore che vogliamo minimizzare, utilizzando una funzione di costo che descriveremo più avanti.
3. Propaghiamo all'indietro l'errore, troviamo la sua derivata rispetto a ciascun peso della rete e aggiorniamo il modello.

Infine, dopo aver ripetuto i passi per più epoch e aver appreso i pesi del modello MLP, utilizziamo la propagazione in avanti per calcolare l'output della rete e applicare una funzione di soglia per ottenere l'etichetta prevista nella rappresentazione one-hot, che abbiamo descritto nel paragrafo precedente.

Ora, attraversiamo i singoli passi della propagazione in avanti per generare un output dagli schemi dei dati di apprendimento. Poiché ciascuna unità della unità nascosta è connessa a tutte le unità dei livelli di input, innanzitutto calcoliamo l'attivazione $a_1^{(2)}$ nel seguente modo:

$$z_1^{(2)} = a_0^{(1)} w_{1,0}^{(1)} + a_1^{(1)} w_{1,1}^{(1)} + \cdots + a_m^{(1)} w_{1,m}^{(1)}$$

$$a_1^{(2)} = \phi\left(z_1^{(2)}\right)$$

Qui, $z_1^{(2)}$ è l'input netto e $\phi(\cdot)$ è la funzione di attivazione, che deve essere differenziabile per determinare i pesi che connettono i neuroni utilizzando un approccio basato su gradienti. Per poter risolvere problemi complessi come la classificazione delle immagini, abbiamo bisogno di funzioni di attivazione non lineare nel nostro modello MLP, per esempio la funzione di attivazione *sigmoid* (*logistica*) che abbiamo utilizzato nella *regressione logistica* del Capitolo 3, *I classificatori di machine learning di scikit-learn*:

$$\phi(z) = \frac{1}{1+e^{-z}}$$

Come ricorderemo, la funzione sigmoid è una curva a "S" che mappa l'input della rete *z* in una distribuzione logistica nell'intervallo compreso fra 0 e 1, che incrocia l'asse *y* in *z=0*, come si può vedere nella Figura 12.4.

Il MLP è un tipico esempio di rete neurale artificiale ad avanzamento (*feedforward*). Il termine *feedforward* fa riferimento al fatto che ciascun livello

funge da input del livello successivo, senza cicli, al contrario di quanto avviene con le reti neurali ricorrenti, un'architettura di cui parleremo più avanti in questo capitolo. Il termine perceptron multilivello può sembrare un po' confuso, in quanto i neuroni artificiali di questa architettura di rete sono tipicamente unità sigmoid e non *perceptron*. Intuitivamente, possiamo considerare i neuroni di un MLP come unità a regressione logistica che restituiscono valori in un intervallo continuo compreso fra 0 e 1.

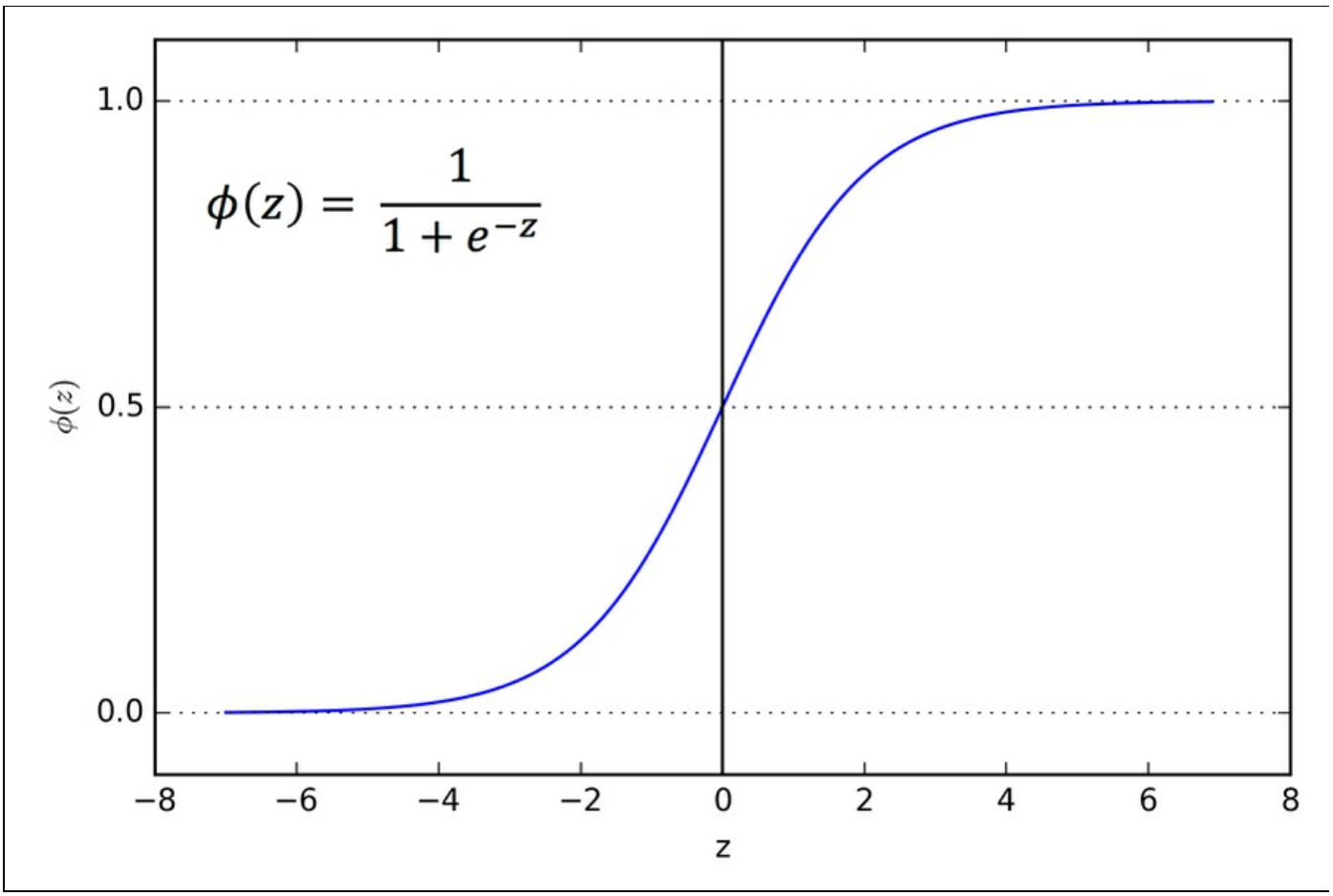


Figura 12.4

Per motivi di efficienza del codice e leggibilità, scriveremo ora l'attivazione in una forma più compatta, utilizzando i concetti dell'algebra lineare semplice, il che ci consentirà di realizzare la nostra implementazione di codice tramite NumPy, invece di scrivere più cicli for Python, nidificati e costosi:

$$\mathbf{z}^{(2)} = \mathbf{W}^{(1)}\mathbf{a}^{(1)}$$

$$\mathbf{a}^{(2)} = \phi\left(\mathbf{z}^{(2)}\right)$$

Qui, $\boldsymbol{a}^{(1)}$ è il nostro vettore delle caratteristiche $[m+1]\times 1$-dimensionale di un campione $\boldsymbol{x}^{(i)}$ più l'unità di bias. $\boldsymbol{W}^{(1)}$ è una matrice dei pesi $h\times[m+1]$-dimensionale, dove *h* è il numero di unità nascoste nella rete neurale. Dopo la moltiplicazione matrice per vettore, otteniamo il vettore di input della rete $h\times 1$-dimensionale $\boldsymbol{z}^{(2)}$ per calcolare l'attivazione $\boldsymbol{a}^{(2)}$ (dove $\boldsymbol{a}^{(2)} \in \mathbb{R}^{h\times 1}$). Inoltre, possiamo generalizzare questo calcolo a tutti gli *n* campioni del set di addestramento:

$$\boldsymbol{Z}^{(2)} = \boldsymbol{W}^{(1)}\left[\boldsymbol{A}^{(1)}\right]^{T}$$

Qui, $\boldsymbol{A}^{(1)}$ è una matrice $n\times[m+1]$ e la moltiplicazione fra matrice e matrice produrrà una matrice di input della rete $h\times n$-dimensionale $\boldsymbol{Z}^{(2)}$. Infine, applichiamo la funzione di attivazione $\phi(\cdot)$ a ciascun valore della matrice di input della rete per ottenere la matrice d'attivazione $h\times n$ per il livello successivo (qui il livello di output):

$$\boldsymbol{A}^{(2)} = \phi\left(\boldsymbol{Z}^{(2)}\right)$$

Analogamente, possiamo riscrivere l'attivazione del livello di output nella forma vettorializzata:

$$\boldsymbol{Z}^{(3)} = \boldsymbol{W}^{(2)}\boldsymbol{A}^{(2)}$$

Qui moltiplichiamo la matrice $\boldsymbol{W}^{(2)}$ di dimensioni $t\times h$ (dove *t* è il numero di unità di output) per la matrice $h\times n$-dimensionale $\boldsymbol{A}^{(2)}$ per ottenere la matrice $t\times n$-dimensionale $\boldsymbol{Z}^{(3)}$ (le colonne di questa matrice rappresentano gli output per ciascun campione).

Infine, applichiamo la funzione di attivazione sigmoid per ottenere l'output a valori continui della nostra rete:

$$\boldsymbol{A}^{(3)} = \phi\left(\boldsymbol{Z}^{(3)}\right),\ \boldsymbol{A}^{(3)} \in \mathbb{R}^{t\times n}$$

Classificazione di cifre scritte a mano

Nel paragrafo precedente, abbiamo introdotto molta teoria relativa all'uso delle reti neurali, ma il tutto può sembrare fin troppo astratto, specialmente per chi affronta per la prima volta questo argomento. Prima di procedere con la discussione dell'algoritmo per l'apprendimento dei pesi del modello MLP, ovvero la retropropagazione, prendiamoci una pausa dal mondo della teoria e vediamo all'opera una rete neurale.

NOTA

La teoria delle reti neurali può essere piuttosto complessa e pertanto è consigliabile dotarsi di due risorse aggiuntive che trattano alcuni dei concetti ai quali abbiamo solo accennato in queste pagine: T. Hastie, J. Friedman e R. Tibshirani, *The Elements of Statistical Learning*, Volume 2, Springer, New York, 2009 e C. M. Bishop et al, *Pattern Recognition and Machine Learning*, Volume 1, Springer, New York, 2006.

Nel prossimo paragrafo descriveremo la nostra prima rete neurale multilivello per classificare le cifre scritte a mano del noto dataset *MNIST* (*Mixed National Institute of Standards and Technology*) che è stato costruito da Yann LeCun et al. e funge da dataset di benchmark per gli algoritmi di apprendimento automatico (Y. LeCun, L. Bottou, Y. Bengio e P. Haffner, *Gradient-based Learning Applied to Document Recognition*, in "Proceedings of the IEEE", 86(11):2278-2324, Novembre 1998).

Procurarsi il dataset MNIST

Il dataset MNIST è liberamente disponibile all'indirizzo http://yann.lecun.com/exdb/mnist/ ed è costituito dalle seguenti quattro parti.

- *Immagini del dataset di apprendimento:* train-images-idx3-ubyte.gz (9.9 MB, 47 MB espanso e 60.000 esempi).
- *Etichette del dataset di apprendimento:* train-labels-idx1-ubyte.gz (29 KB, 60 KB espanso e 60.000 etichette).
- *Immagini del dataset di test:* t10k-images-idx3-ubyte.gz (1.6 MB, 7.8 MB espanso e 10.000 esempi).
- *Etichette del dataset di test:* t10k-labels-idx1-ubyte.gz (5 KB, 10 KB espanso e 10.000 etichette).

Il dataset MNIST è stato costruito a partire da due dataset del *NIST* (*National Institute of Standards and Technology*) statunitense. Il set di addestramento è

costituito da cifre scritte a mano da 250 persone differenti, per la metà studenti delle scuole superiori e per la metà dipendenti del Census Bureau. Notate che il set di test contiene cifre scritte a mano da persone differenti, che seguono lo stesso criterio di suddivisione.

Dopo aver scaricato i file, è opportuno espanderli utilizzando lo strumento Unix/Linux `gzip` dalla riga di comando del Terminale, utilizzando il seguente comando dopo aver fatto accesso alla directory locale nella quale avete scaricato il database:

```
gzip *ubyte.gz -d
```

Alternativamente, potreste utilizzare uno strumento di espansione a parte, per esempio se impiegate una macchina Windows. Le immagini sono conservate in formato byte e le leggeremo all'interno di array NumPy che utilizzeremo poi per addestrare e collaudare la nostra implementazione di un MLP:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte'
                                % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte'
                               % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)
    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                 dtype=np.uint8).reshape(len(labels), 784)

    return images, labels
```

La funzione `load_mnist` restituisce due array: il primo (`images`) è un array NumPy $n \times m$-dimensionale, dove n è il numero dei campioni e mè il numero delle caratteristiche. Il dataset di addestramento è costituito da 60.000 cifre, mentre il dataset di test contiene 10.000 campioni. Le immagini contenute nel dataset MNIST sono formate da 28 × 28 pixel e ogni pixel è rappresentato da un valore di intensità in scala di grigio. Qui, espandiamo i 28 × 28 pixel in vettori riga monodimensionali, che rappresentano le righe nel nostro array di immagini (784 per riga o immagine). Il secondo array (`labels`) restituito dalla funzione `load_mnist` contiene la corrispondente variabile target, le etichette delle classi (numeri interi compresi fra 0 e 9) delle cifre

scritte a mano.

Il modo in cui leggiamo l'immagine può sembrare a prima vista piuttosto strano:

```
magic, n = struct.unpack('>II', lbpath.read(8))
labels = np.fromfile(lbpath, dtype=np.int8)
```

Per comprendere il funzionamento di queste due righe di codice, diamo un'occhiata alla descrizione del set dal sito web MNIST:

```
[offset] [type] [value] [description]
0000 32 bit integer 0x00000801(2049) magic number (MSB first)
0004 32 bit integer 60000 number of items
0008 unsigned byte ?? label
0009 unsigned byte ?? label
........
xxxx unsigned byte ?? label
```

Utilizzando le due righe di codice precedenti, leggiamo innanzitutto il *magic number*, che descrive il protocollo del file e poi il *numero di elementi* (*n*) dal buffer del file prima di leggere i byte successivi in un array NumPy utilizzando il metodo `fromfile`. Il valore `>II` del parametro `fmt` che abbiamo passato come argomento a `struct.unpack` ha due parti.

- `>`: significa *big-endian* (definisce l'ordine in cui è conservata una sequenza di byte); se non conoscete i termini *big-endian* e *small-endian*, trovate un eccellente articolo su Wikipedia (https://it.wikipedia.org/wiki/Ordine_dei_byte).
- `I`: significa intero senza segno (*unsigned integer*).

Tramite il codice seguente, caricheremo le 60.000 istanze di addestramento e anche i 10.000 campioni di test dalla directory `mnist` nella quale abbiamo espanso il dataset MNIST:

```
>>> X_train, y_train = load_mnist('mnist', kind='train')
>>> print('Rows: %d, columns: %d'
...       % (X_train.shape[0], X_train.shape[1]))
Rows: 60000, columns: 784
>>> X_test, y_test = load_mnist('mnist', kind='t10k')
>>> print('Rows: %d, columns: %d'
...       % (X_test.shape[0], X_test.shape[1]))
Rows: 10000, columns: 784
```

Per avere un'idea dell'aspetto delle immagini, visualizzeremo gli esempi delle cifre comprese fra 0 e 9 dopo aver recuperato la forma dei vettori di 784 pixel estraendoli dalla matrice delle caratteristiche e assegnando loro le dimensioni originali 28 × 28 che possiamo rappresentare tramite la funzione `imshow` di matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X_train[y_train == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys', interpolation='nearest')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

Dovremmo vedere un grafico di 2×5 piccole figure, che mostrano un campione rappresentativo di ciascuna cifra (Figura 12.5).

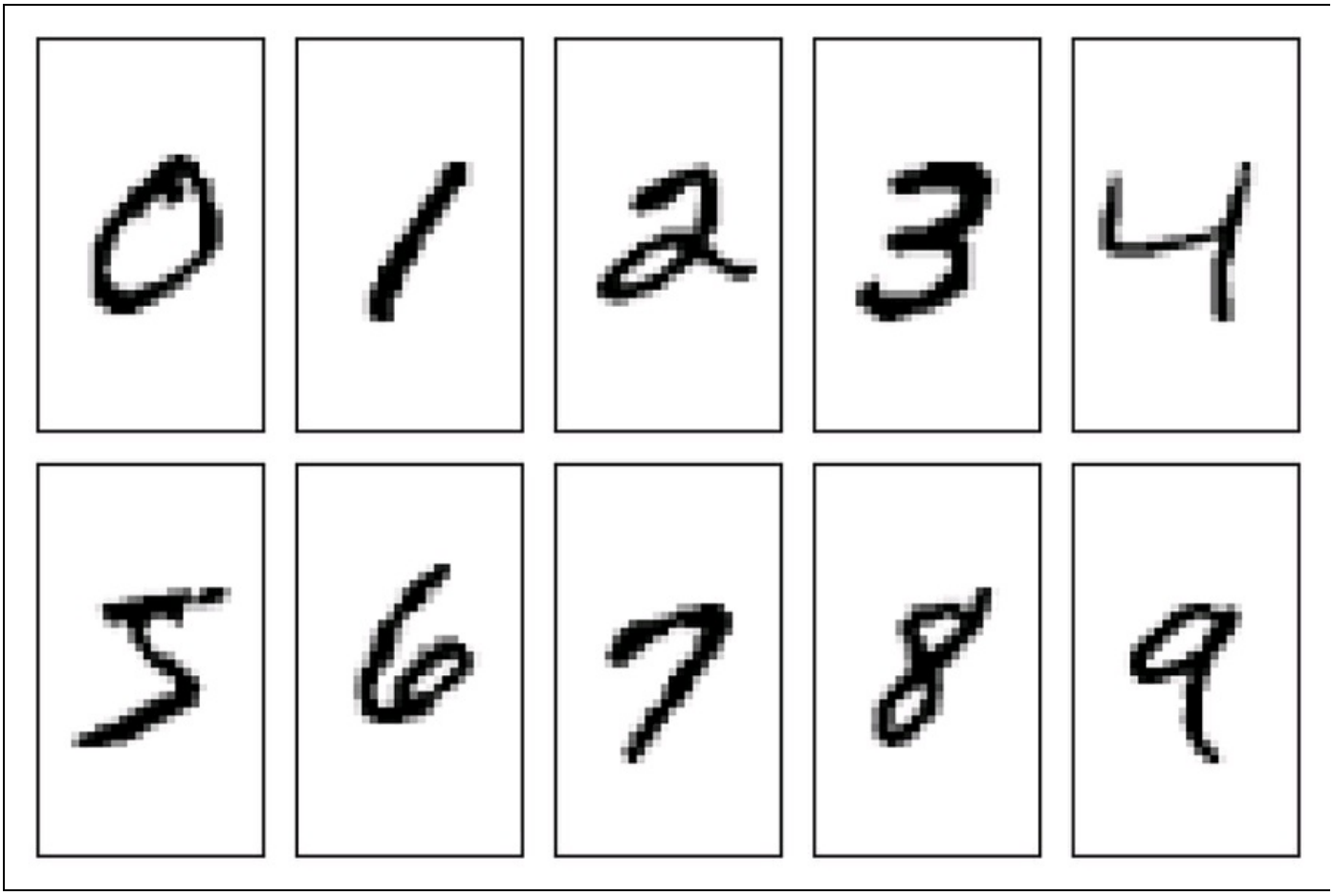


Figura 12.5

Inoltre, tracciamo anche degli esempi della stessa cifra, per vedere quale può essere il loro aspetto:

```
>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = X_train[y_train == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys', interpolation='nearest')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

Dopo l'esecuzione del codice, troveremo 25 varianti della cifra, il "7" (Figura 12.6).

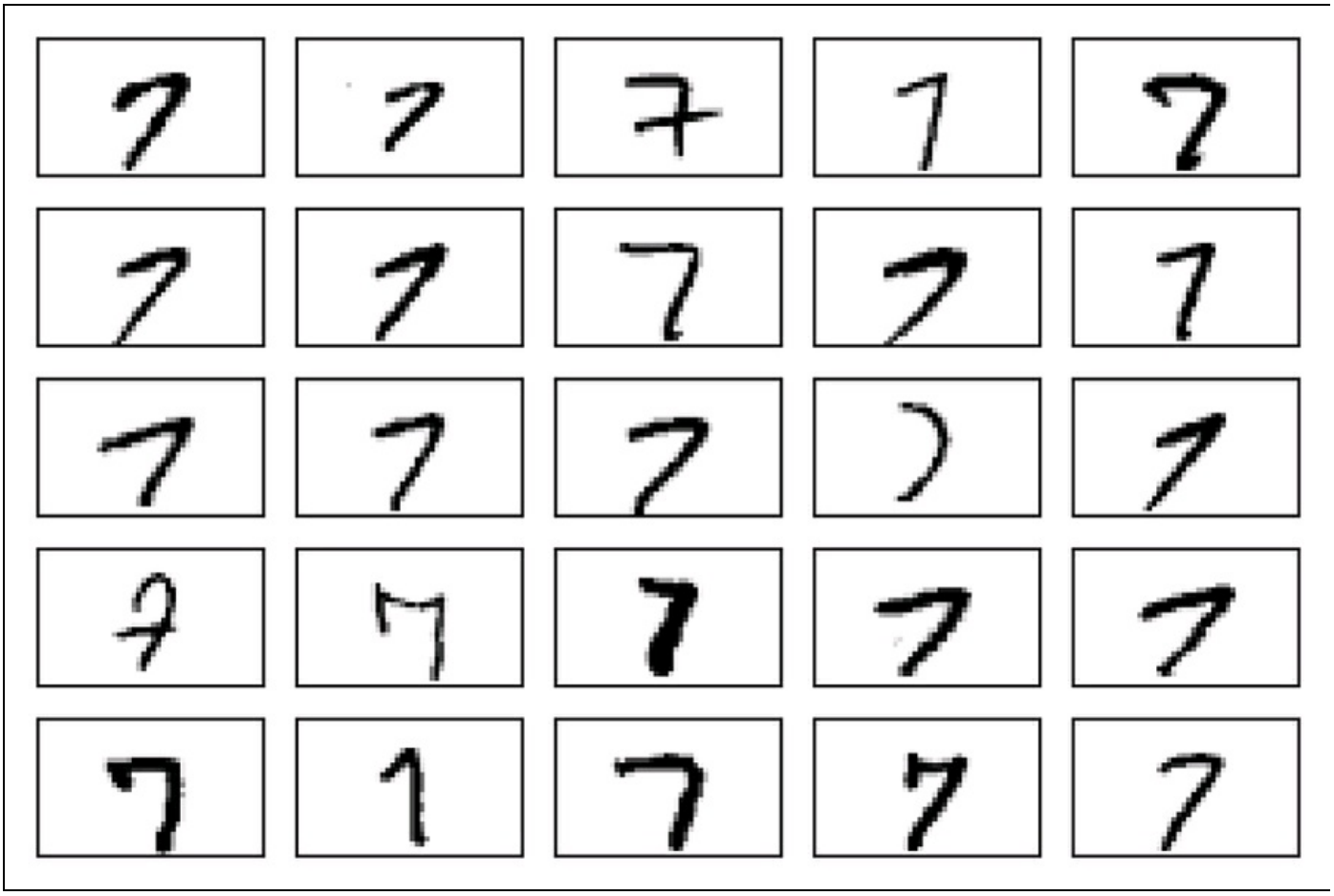


Figura 12.6

Opzionalmente, possiamo salvare i dati e le etichette delle immagini MNIST sotto forma di file CSV, per poi aprirli all'interno di programmi che non supportano il formato a byte. Tuttavia, dobbiamo anche considerare che il formato di file CSV occuperà molto più spazio sull'unità dischi locale, come si può vedere di seguito.

- train_img.csv: 109.5 MB.
- train_labels.csv: 120 KB.
- test_img.csv: 18.3 MB.
- test_labels: 20 KB.

Se decidiamo di salvare questi file, possiamo eseguire il seguente codice nella nostra sessione Python dopo aver caricato i dati MNIST negli array NumPy:

```
>>> np.savetxt('train_img.csv', X_train,
...            fmt='%i', delimiter=',')
>>> np.savetxt('train_labels.csv', y_train,
...            fmt='%i', delimiter=',')
>>> np.savetxt('test_img.csv', X_test,
...            fmt='%i', delimiter=',')
>>> np.savetxt('test_labels.csv', y_test,
...            fmt='%i', delimiter=',')
```

Dopo aver salvato i file CSV, possiamo caricarli di nuovo in Python utilizzando la funzione genfromtxt di NumPy:

```
>>> X_train = np.genfromtxt('train_img.csv',
...                         dtype=int, delimiter=',')
>>> y_train = np.genfromtxt('train_labels.csv',
...                         dtype=int, delimiter=',')
>>> X_test = np.genfromtxt('test_img.csv',
...                        dtype=int, delimiter=',')
>>> y_test = np.genfromtxt('test_labels.csv',
...                        dtype=int, delimiter=',')
```

Tuttavia, richiede parecchio più tempo caricare i dati MNIST dai file CSV; pertanto, se possibile, si consiglia di utilizzare il formato originale, byte.

Implementazione di un perceptron multilivello

In questo paragrafo implementeremo il codice di un MLP con un livello di input, un livello nascosto e un livello di output, per classificare le immagini contenute nel dataset MNIST. Abbiamo fatto ogni tentativo di mantenere il più possibile semplice il codice. Tuttavia, inizialmente può sembrare un po' complicato e per questo sarebbe opportuno scaricare il codice d'esempio di questo capitolo, dove si può trovare un'implementazione del MLP molto più ricca di commenti e di evidenziazioni sintattiche, per migliorarne la leggibilità. Se non eseguite il codice dal notebook IPython di accompagnamento, è consigliabile copiarlo in un file script Python all'interno della directory di lavoro corrente, per esempio in neuralnet.py; poi potrete importarlo nella sessione Python tramite il seguente comando:

```
from neuralnet import NeuralNetMLP
```

Il codice conterrà parti di cui non abbiamo ancora parlato, come l'algoritmo di retro-propagazione, ma la maggior parte del codice dovrebbe risultare familiare sulla base dell'implementazione di Adaline presentata nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione* e della discussione sulla propagazione in avanti dei paragrafi precedenti. Non preoccupatevi se non tutto il codice sembra essere immediatamente comprensibile; descriveremo alcune sue parti nel corso del capitolo. Tuttavia, il fatto di esaminare il codice in questa fase del capitolo faciliterà la comprensione della teoria , presentata nelle prossime pagine.

```
import numpy as np
from scipy.special import expit
import sys
class NeuralNetMLP(object):
    def __init__(self, n_output, n_features, n_hidden=30,
                 l1=0.0, l2=0.0, epochs=500, eta=0.001,
                 alpha=0.0, decrease_const=0.0, shuffle=True,
                 minibatches=1, random_state=None):
        np.random.seed(random_state)
        self.n_output = n_output
        self.n_features = n_features
        self.n_hidden = n_hidden
        self.w1, self.w2 = self._initialize_weights()
```

```
    self.l1 = l1
    self.l2 = l2
    self.epochs = epochs
    self.eta = eta
    self.alpha = alpha
    self.decrease_const = decrease_const
    self.shuffle = shuffle
    self.minibatches = minibatches

def _encode_labels(self, y, k):
    onehot = np.zeros((k, y.shape[0]))
    for idx, val in enumerate(y):
        onehot[val, idx] = 1.0
    return onehot

def _initialize_weights(self):
    w1 = np.random.uniform(-1.0, 1.0,
                size=self.n_hidden*(self.n_features + 1))
    w1 = w1.reshape(self.n_hidden, self.n_features + 1)
    w2 = np.random.uniform(-1.0, 1.0,
                size=self.n_output*(self.n_hidden + 1))
    w2 = w2.reshape(self.n_output, self.n_hidden + 1)
    return w1, w2

def _sigmoid(self, z):
    # expit is equivalent to 1.0/(1.0 + np.exp(-z))
    return expit(z)

def _sigmoid_gradient(self, z):
    sg = self._sigmoid(z)
    return sg * (1 - sg)

def _add_bias_unit(self, X, how='column'):
    if how == 'column':
        X_new = np.ones((X.shape[0], X.shape[1]+1))
        X_new[:, 1:] = X
    elif how == 'row':
        X_new = np.ones((X.shape[0]+1, X.shape[1]))
        X_new[1:, :] = X
    else:
        raise AttributeError('`how` must be `column` or `row`')
    return X_new

def _feedforward(self, X, w1, w2):
    a1 = self._add_bias_unit(X, how='column')
    z2 = w1.dot(a1.T)
    a2 = self._sigmoid(z2)
    a2 = self._add_bias_unit(a2, how='row')
    z3 = w2.dot(a2)
    a3 = self._sigmoid(z3)
    return a1, z2, a2, z3, a3

def _L2_reg(self, lambda_, w1, w2):
    return (lambda_/2.0) * (np.sum(w1[:, 1:] ** 2)\
          + np.sum(w2[:, 1:] ** 2))

def _L1_reg(self, lambda_, w1, w2):
    return (lambda_/2.0) * (np.abs(w1[:, 1:]).sum()\
          + np.abs(w2[:, 1:]).sum())

def _get_cost(self, y_enc, output, w1, w2):
    term1 = -y_enc * (np.log(output))
    term2 = (1 - y_enc) * np.log(1 - output)
    cost = np.sum(term1 - term2)
    L1_term = self._L1_reg(self.l1, w1, w2)
    L2_term = self._L2_reg(self.l2, w1, w2)
    cost = cost + L1_term + L2_term
    return cost

def _get_gradient(self, a1, a2, a3, z2, y_enc, w1, w2):
    # backpropagation
    sigma3 = a3 - y_enc
    z2 = self._add_bias_unit(z2, how='row')
    sigma2 = w2.T.dot(sigma3) * self._sigmoid_gradient(z2)
    sigma2 = sigma2[1:, :]
    grad1 = sigma2.dot(a1)
    grad2 = sigma3.dot(a2.T)
```

```
        # regularize
        grad1[:, 1:] += (w1[:, 1:] * (self.l1 + self.l2))
        grad2[:, 1:] += (w2[:, 1:] * (self.l1 + self.l2))
        return grad1, grad2

    def predict(self, X):
        a1, z2, a2, z3, a3 = self._feedforward(X, self.w1, self.w2)
        y_pred = np.argmax(z3, axis=0)
        return y_pred

    def fit(self, X, y, print_progress=False):
        self.cost_ = []
        X_data, y_data = X.copy(), y.copy()
        y_enc = self._encode_labels(y, self.n_output)
        delta_w1_prev = np.zeros(self.w1.shape)
        delta_w2_prev = np.zeros(self.w2.shape)
        for i in range(self.epochs):
            # adaptive learning rate
            self.eta /= (1 + self.decrease_const*i)
            if print_progress:
                sys.stderr.write(
                      '\rEpoch: %d/%d' % (i+1, self.epochs))
                sys.stderr.flush()
            if self.shuffle:
                idx = np.random.permutation(y_data.shape[0])
                X_data, y_enc = X_data[idx], y_enc[:,idx]
            mini = np.array_split(range(
                    y_data.shape[0]), self.minibatches)
            for idx in mini:
                # feedforward
                a1, z2, a2, z3, a3 = self._feedforward(
                                X_data[idx], self.w1, self.w2)
                cost = self._get_cost(y_enc=y_enc[:, idx],
                                output=a3,
                                w1=self.w1,
                                w2=self.w2)
                self.cost_.append(cost)              # compute gradient via backpropagation
                grad1, grad2 = self._get_gradient(a1=a1, a2=a2,
                                    a3=a3, z2=z2,
                                    y_enc=y_enc[:, idx],
                                    w1=self.w1,
                                    w2=self.w2)
                # update weights
                delta_w1, delta_w2 = self.eta * grad1,\
                                self.eta * grad2
                self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
                self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
                delta_w1_prev, delta_w2_prev = delta_w1, delta_w2
        return self
```

Ora, inizializziamo un nuovo MLP 784-50-10, ovvero una rete neurale con 784 unità di input (n_features), 50 unità nascoste (n_hidden) e 10 unità di output (n_output):

```
>>> nn = NeuralNetMLP(n_output=10,
... n_features=X_train.shape[1],
... n_hidden=50,
... l2=0.1,
... l1=0.0,
... epochs=1000,
... eta=0.001,
... alpha=0.001,
... decrease_const=0.00001,
... shuffle=True,
... minibatches=50,
... random_state=1)
```

Come forse avrete notato, sulla base della nostra precedente implementazione del MLP, abbiamo anche implementato alcune funzionalità aggiuntive, fra cui le seguenti.

- l2: il parametro λ per la regolarizzazione L2, per ridurre il grado di overfitting; analogamente, l1 è il parametro λper la regolarizzazione L1.
- epochs: il numero di passi nel corso del set di addestramento.
- eta: il tasso di apprendimento η.
- alpha: il parametro *momentum learning*, per aggiungere all'aggiornamento del peso un fattore del gradiente precedente, in modo da accelerare l'apprendimento $\Delta w_t = \eta \nabla J(w_t) + \alpha \Delta w_{t-1}$ (dove t è il passo corrente o epoch).
- decrease_const: la costante di decremento *d* per un tasso di apprendimento adattativo n che si riduce nel corso del tempo per migliorare la convergenza $\eta / 1 + t \times d$.
- shuffle: si mescola il set di apprendimento prima di ogni epoch in modo da evitare che l'algoritmo si trovi a operare all'interno di cicli.
- Minibatches: la suddivisione dei dati di addestramento in *k* mini-lotti a ogni epoch. Il gradiente viene calcolato separatamente per ogni mini-lotto invece che per l'intero dataset di addestramento, in modo da accelerare l'apprendimento.

Poi, addestriamo il MLP utilizzando i 60.000 campioni del dataset di addestramento MNIST già mescolati. Prima di eseguire il codice seguente, considerate che l'addestramento della rete neurale può richiedere dai 10 ai 30 minuti su un comune computer desktop:

```
>>> nn.fit(X_train, y_train, print_progress=True)
Epoch: 1000/1000
```

Analogamente alla precedente implementazione di Adaline, abbiamo salvato il costo per ogni epoch in una lista cost_ che ora possiamo visualizzare, assicurandoci che l'algoritmo di ottimizzazione abbia raggiunto la convergenza. Qui, tracciamo solo ogni cinquantesimo passo, per considerare i 50 mini-lotti (50 mini-lotti per 1000 epoch). Il codice è il seguente:

```
>>> plt.plot(range(len(nn.cost_)), nn.cost_)
>>> plt.ylim([0, 2000])
>>> plt.ylabel('Cost')
>>> plt.xlabel('Epochs * 50')
>>> plt.tight_layout()
>>> plt.show()
```

Come possiamo vedere nella Figura 12.7, il grafico della funzione di costo sembra affetto da un certo rumore. Questo è dovuto al fatto che abbiamo addestrato la nostra rete neurale con un apprendimento mini-batch, una variante della discesa del gradiente stocastica.

Sebbene possiamo già vedere dal grafico che l'algoritmo di ottimizzazione converge dopo circa 800 epoch (*40.000/50 = 800*), tracciamo una versione più

arrotondata della funzione di costo rispetto al numero di epoch, calcolando la media rispetto agli intervalli dei mini-lotti. Il codice è il seguente:

```
>>> batches = np.array_split(range(len(nn.cost_)), 1000)
>>> cost_ary = np.array(nn.cost_)
>>> cost_avgs = [np.mean(cost_ary[i]) for i in batches]
>>> plt.plot(range(len(cost_avgs)),
...          cost_avgs,
...          color='red')
>>> plt.ylim([0, 2000])
>>> plt.ylabel('Cost')
>>> plt.xlabel('Epochs')
>>> plt.tight_layout()
>>> plt.show()
```

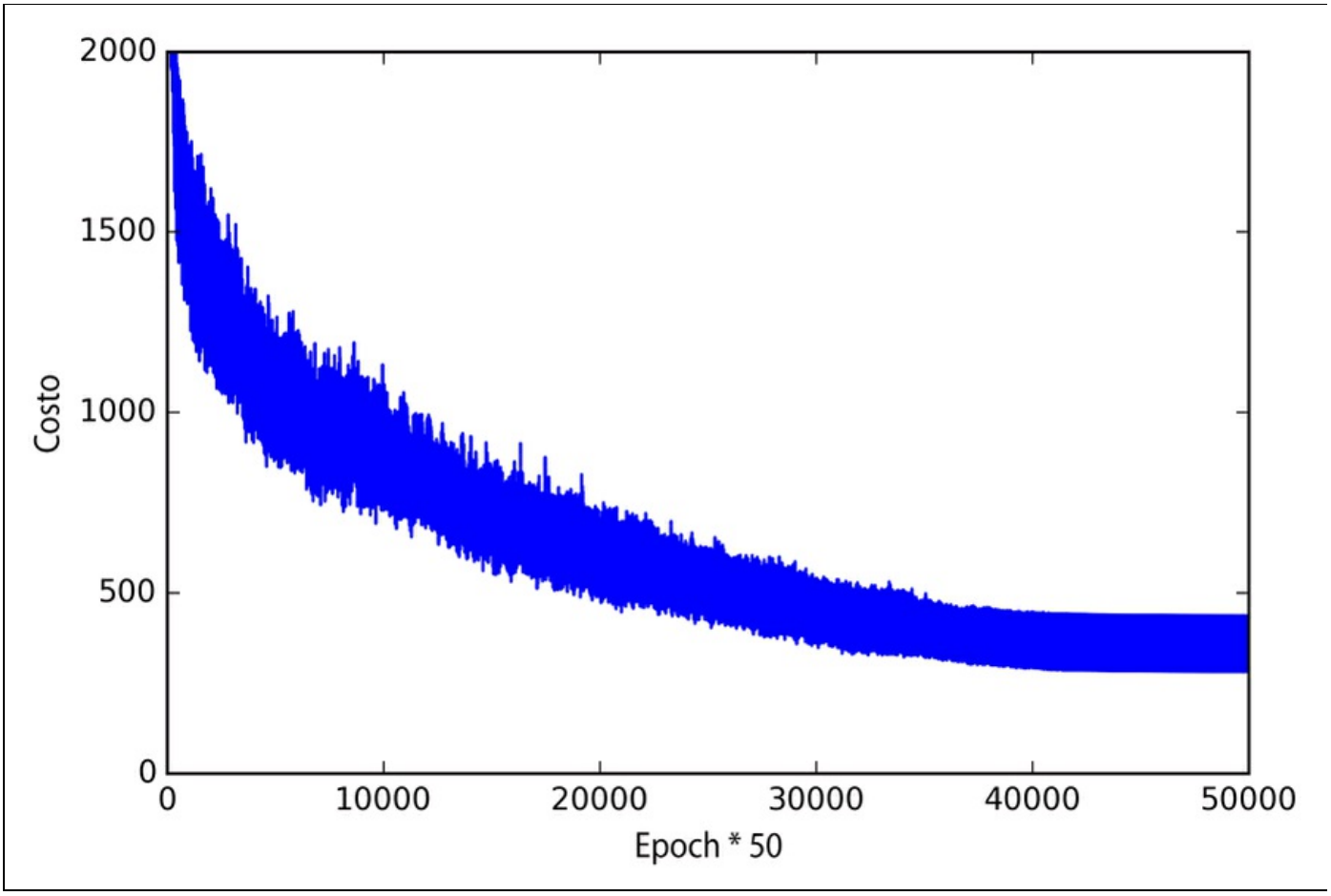


Figura 12.7

Il grafico della Figura 12.8 ci dà una rappresentazione più chiara, che indica che l'algoritmo di addestramento ha iniziato a convergere poco dopo l'epoch numero 800.

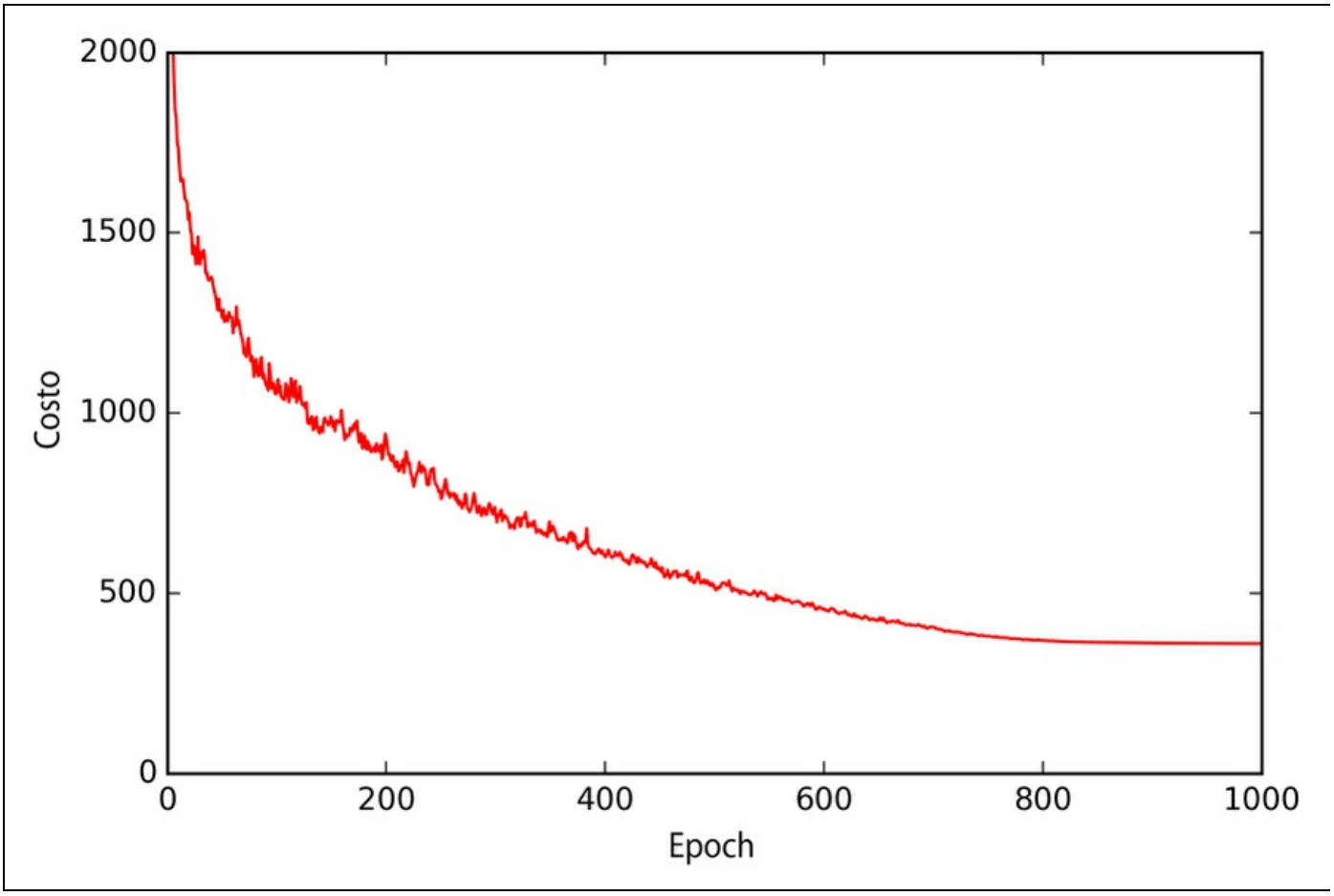


Figura 12.8

Ora, valutiamo le prestazioni del modello, calcolando l'accuratezza della previsione:

```
>>> y_train_pred = nn.predict(X_train)
>>> acc = np.sum(y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Training accuracy: %.2f%%' % (acc * 100))
Training accuracy: 97.59%
```

Come possiamo vedere, il modello classifica correttamente la maggior parte delle cifre, ma questi risultati possono essere generalizzati su dati che il modello non ha mai visto prima? Calcoliamo l'accuratezza sulle 10.000 immagini che appartengono al dataset di test:

```
>>> y_test_pred = nn.predict(X_test)
>>> acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
>>> print('Test accuracy: %.2f%%' % (acc * 100))
Test accuracy: 95.62%
```

Sulla base della discrepanza esistente fra l'accuratezza di addestramento e di test, possiamo concludere che il modello soffre di un leggero overfit sui dati di addestramento. Per ottimizzare ulteriormente il modello, potremmo intervenire sul numero di unità nascoste, sui valori dei parametri di regolarizzazione, sul tasso di apprendimento, sui valori della costante di decrescita o sull'apprendimento adattativo, utilizzando le tecniche di cui abbiamo parlato nel Capitolo 6,

Valutazione dei modelli e ottimizzazione degli iperparametri (questo viene lasciato come esercizio per il lettore).

Ora, diamo un'occhiata ad alcune delle immagini che risultano più problematiche per il nostro MLP:

```
>>> miscl_img = X_test[y_test != y_test_pred][:25]
>>> correct_lab = y_test[y_test != y_test_pred][:25]
>>> miscl_lab= y_test_pred[y_test != y_test_pred][:25]
>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = miscl_img[i].reshape(28, 28)
...     ax[i].imshow(img,
...                  cmap='Greys',
...                  interpolation='nearest')
...     ax[i].set_title('%d) t: %d p: %d'
...                     % (i+1, correct_lab[i], miscl_lab[i]))
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

Dovremmo vedere (Figura 12.9) una matrice di immagini 5×5, dove il primo numero indica la posizione nel grafico, il secondo indica l'etichetta della classe vera (t) e il terzo indica invece l'etichetta della classe prevista (p).

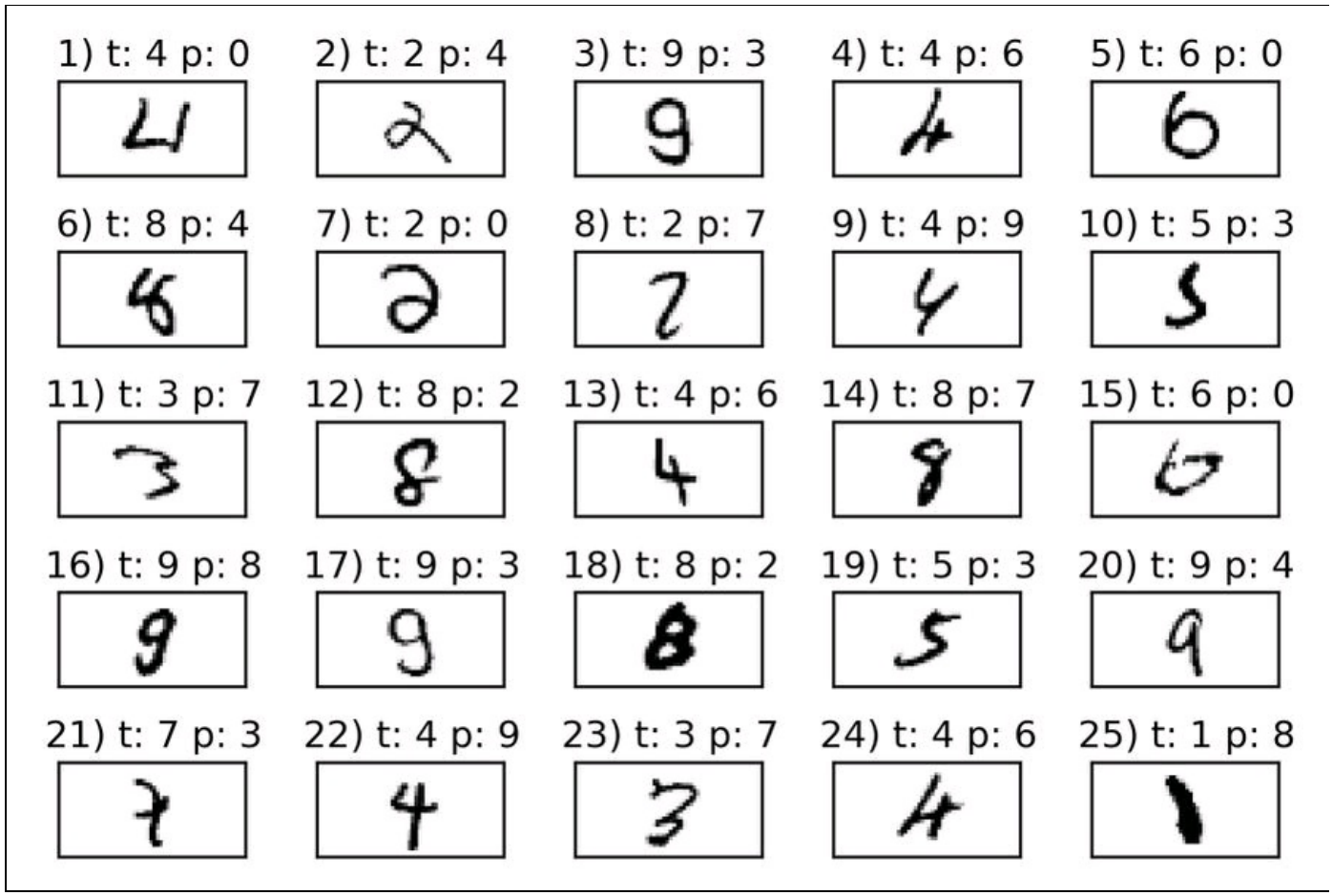


Figura 12.9

Come possiamo vedere dalla Figura 12.9, alcune di queste immagini sarebbero difficilmente classificabili anche da una persona umana. Per esempio, possiamo vedere che la cifra “9” (immagini numero 3, 16 e 17) può essere classificata come un “3” o come un “8” a seconda del fatto che la parte inferiore della cifra abbia una curvatura più o meno pronunciata.

Addestramento di una rete neurale artificiale

Ora che abbiamo visto in azione una rete neurale e sappiamo qualcosa di più del suo funzionamento avendo esaminato il codice, entriamo più in profondità in alcuni dei suoi concetti, come la funzione di costo logistica e l'algoritmo di retropropagazione che abbiamo implementato per individuare i pesi.

Calcolare la funzione logistica di costo

La funzione logistica di costo che abbiamo implementato con il metodo `_get_cost` è in realtà piuttosto semplice da interpretare, poiché si tratta della stessa funzione di costo che abbiamo descritto nella sezione dedicata alla regressione logistica nel Capitolo 3, *I classificatori di machine learning di scikit-learn*.

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)$$

Qui, $a^{(i)}$ è un'attivazione della *i*-esima unità in uno dei livelli che calcoliamo nel passo di propagazione in avanti:

$$a^{(i)} = \phi\left(z^{(i)}\right)$$

Ora, aggiungiamo un termine di *regolarizzazione*, che ci consenta di ridurre il grado di overfitting. Come ricorderete dei capitoli precedenti, i termini di regolarizzazione L2 e L1 sono definiti nel seguente modo (ricordate che non regolarizziamo le unità di bias):

$$L2 = \lambda \left\| \boldsymbol{w} \right\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2 \text{ and } L1 = \lambda \left\| \boldsymbol{w} \right\|_1^1 = \lambda \sum_{j=1}^{m} \left| w_j \right|$$

Sebbene la nostra implementazione del MLP supporti entrambe le regolarizzazioni (L1 e L2), per semplicità ci concentreremo solo sul termine di regolarizzazione L2. Tuttavia valgono gli stessi concetti anche per L1.

Aggiungendo alla nostra funzione il termine di regolarizzazione L2, otteniamo la seguente equazione:

$$J(\boldsymbol{w}) = -\left[\sum_{i=1}^{n} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)\right] + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$$

Poiché abbiamo implementato un MLP per multiclasse, otteniamo in output un vettore di *t* elementi, che dobbiamo confrontare con il vettore target $t \times 1$-dimensionale nella rappresentazione a codifica one-hot. Per esempio, l'attivazione del terzo livello e della classe target (in questo caso la classe 2) per un determinato campione può avere il seguente aspetto:

$$a^{(3)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Pertanto, dobbiamo generalizzare la funzione logistica di costo a tutte le unità di attivazione *j* della nostra rete. Così la nostra funzione di costo (senza il termine di regolarizzazione) diviene:

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{(i)} \log\left(a_j^{(i)}\right) + \left(1 - y_j^{(i)}\right) \log\left(1 - a_j^{(i)}\right)$$

Qui, l'indice *i* fa riferimento a un determinato campione del set di addestramento.

Il seguente termine di regolarizzazione generalizzata può sembrare un po' complicato a prima vista, ma stiamo semplicemente calcolando la somma di tutti i pesi di un livello *l* (senza il termine bias) che abbiamo aggiunto alla prima colonna:

$$J(\boldsymbol{w}) = -\left[\sum_{i=1}^{n}\sum_{j=1}^{m} y_j^{(i)} \log\left(\phi\left(z^{(i)}\right) j\right) + \left(1 - y_j^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right) j\right)\right]$$
$$+ \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2$$

La seguente equazione rappresenta il termine di penalizzazione L2:

$$\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2$$

Ricordate che il nostro obiettivo è minimizzare la funzione di costo $J(\boldsymbol{w})$. Pertanto, dobbiamo calcolare la derivata parziale della matrice $\boldsymbol{W}$ rispetto a ciascun peso per ogni livello della rete:

$$\frac{\partial}{\partial w_{j,i}^{(l)}} J(\boldsymbol{W})$$

Nel prossimo paragrafo, parleremo dell'algoritmo di retropropagazione, che ci consente di calcolare queste derivate parziali in modo da minimizzare la funzione di costo.

Notate che $\boldsymbol{W}$ è costituita da più matrici. In un perceptron multilivello con una unità nascosta, abbiamo la matrice dei pesi $\boldsymbol{W}^{(1)}$, che connette l'input al livello nascosto e $\boldsymbol{W}^{(2)}$ che connette il livello nascosto al livello di output. Una visualizzazione intuitiva della matrice $\boldsymbol{W}$ può essere quella rappresentata nella Figura 12.10.

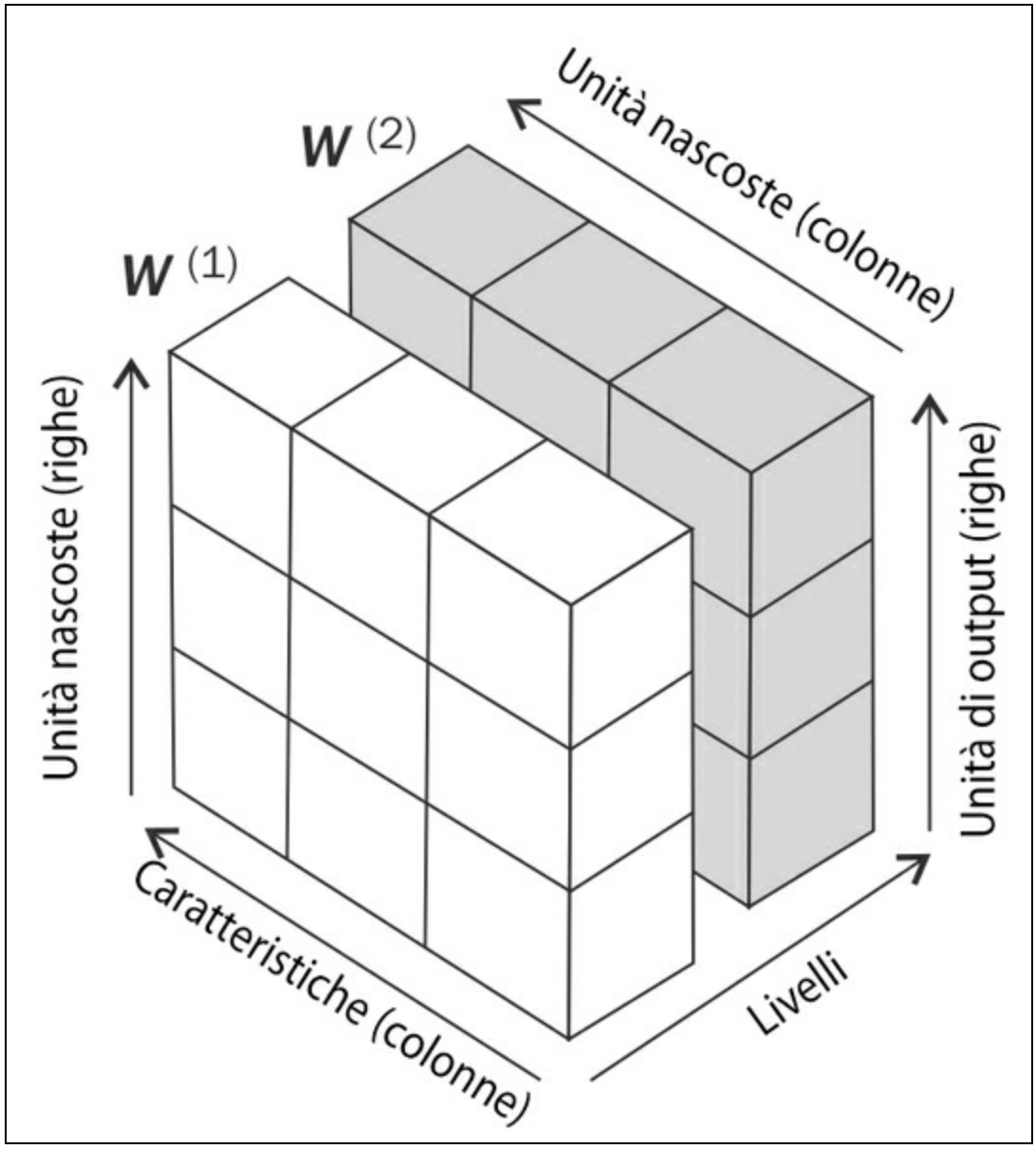


Figura 12.10

In questa figura semplificata, può sembrare che $\boldsymbol{W}^{(1)}$ e $\boldsymbol{W}^{(2)}$ abbiano lo stesso numero di righe e colonne, ma in genere le cose non stanno così, a meno che inizializzate un MLP con lo stesso numero di unità nascoste, unità di output e caratteristiche di input.

Se l'argomento sembra complesso, procedete con la lettura del prossimo paragrafo, dove parleremo più in dettaglio della dimensionalità di $\boldsymbol{W}^{(1)}$ e $\boldsymbol{W}^{(2)}$ nel contesto dell'algoritmo di retropropagazione.

Addestramento delle reti neurali tramite retropropagazione

In questo paragrafo esamineremo gli elementi matematici della retropropagazione, per comprendere come si possono individuare in modo molto efficiente i pesi in una rete neurale. A seconda della competenza matematica del lettore, le seguenti equazioni possono sembrare relativamente complicate a prima vista. Molti preferiscono un approccio *bottom-up*, ed esaminare le equazioni passo dopo passo, in modo da sviluppare progressivamente un'idea del funzionamento degli algoritmi. Tuttavia, se preferite un approccio *top-down* e volete conoscere il funzionamento della retropropagazione senza l'ausilio delle notazioni matematiche, vi consiglio la lettura del prossimo paragrafo, *Aspetti intuitivi della retropropagazione*, per tornare poi a questo paragrafo.

Nel paragrafo precedente, abbiamo visto come si calcola il costo, come la differenza fra l'attivazione dell'ultimo livello e l'etichetta della classe target. Ora, vedremo come funziona l'algoritmo di retropropagazione per aggiornare i pesi del nostro modello MLP, che abbiamo implementato nel metodo `_get_gradient`. Come ricorderete da quanto abbiamo detto all'inizio di questo capitolo, innanzitutto dobbiamo applicare la propagazione in avanti, in modo da ottenere l'attivazione del livello di output, che abbiamo formulato nel seguente modo:

$$\boldsymbol{Z}^{(2)} = \boldsymbol{W}^{(1)}\left[\boldsymbol{A}^{(1)}\right]^{T} \text{ (input di rete del livello nascosto)}$$

$$\boldsymbol{A}^{(2)} = \phi\left(\boldsymbol{Z}^{(2)}\right) \text{ (attivazione del livello nascosto)}$$

$$\boldsymbol{Z}^{(3)} = \boldsymbol{W}^{(2)}\boldsymbol{A}^{(2)} \text{ (input di rete del livello di output)}$$

$$\boldsymbol{A}^{(3)} = \phi\left(\boldsymbol{Z}^{(3)}\right) \text{ (attivazione del livello di output)}$$

In sintesi, propaghiamo solo in avanti le caratteristiche di input, attraverso le connessioni di rete, come illustrato nella Figura 12.11.

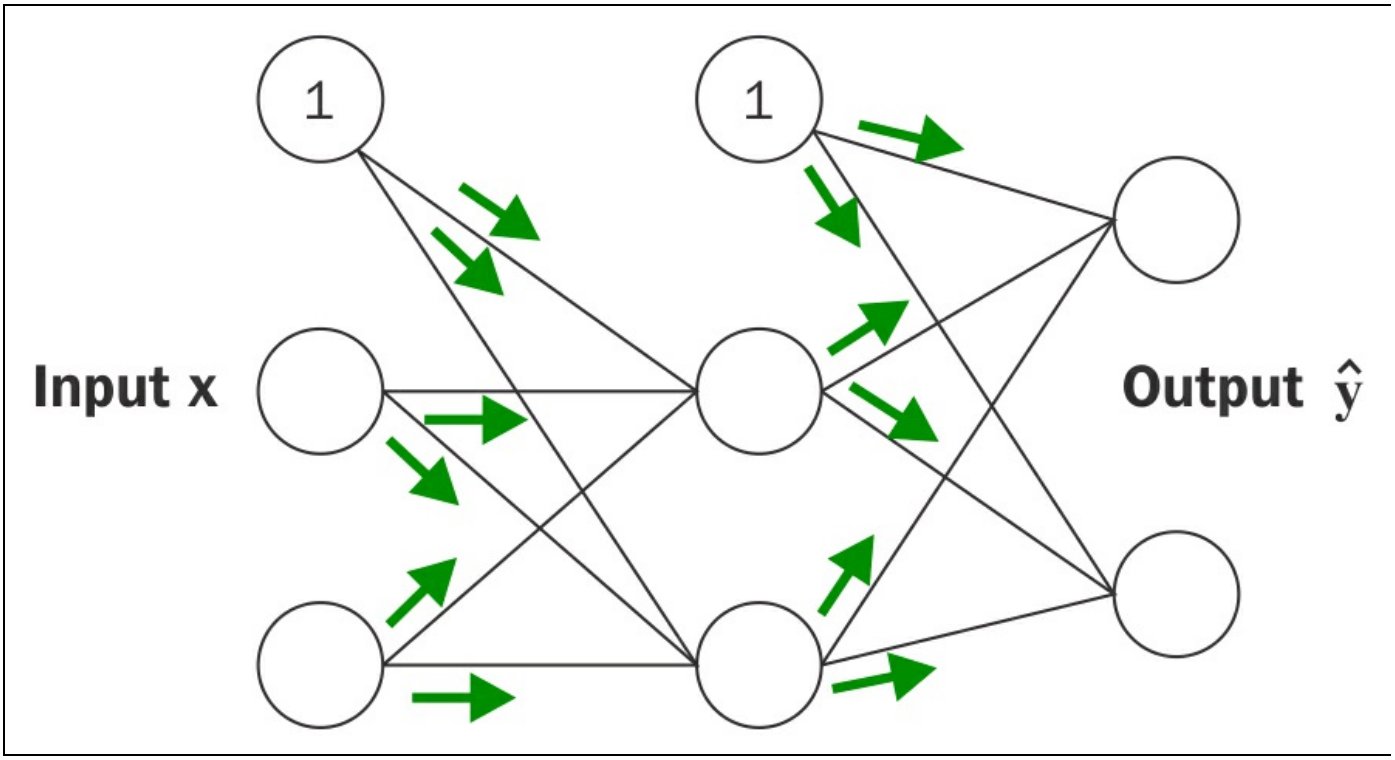


Figura 12.11

Nella retropropagazione, propaghiamo l'errore da destra verso sinistra. Iniziamo calcolando il vettore degli errori del livello di output:

$$\delta^{(3)} = a^{(3)} - y$$

Qui, y è il vettore delle vere etichette delle classi.

Poi calcoliamo il termine d'errore nel livello nascosto:

$$\boldsymbol{\delta}^{(2)} = \left(\boldsymbol{W}^{(2)}\right)^{T} \boldsymbol{\delta}^{(3)} * \frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}}$$

Qui

$$\frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}}$$

è semplicemente la derivata della funzione di attivazione, che abbiamo implementato come `_sigmoid_gradient`:

$$\frac{\partial \phi(z)}{\partial z} = \left(a^{(2)} * \left(1 - a^{(2)}\right)\right)$$

Notate che qui il simbolo di asterisco (*) rappresenta una moltiplicazione elemento per elemento.

Approfondimento

Sebbene non sia importante seguire le prossime equazioni, potreste chiedervi come sia stata ottenuta la derivata della funzione di attivazione. Riepiloghiamo di seguito i passi di derivazione:

$$\phi'(z) = \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right)$$

$$= \frac{e^{-z}}{\left(1+e^{-z}\right)^2}$$

$$= \frac{1+e^{-z}}{\left(1+e^{-z}\right)^2} - \left(\frac{1}{1+e^{-z}}\right)^2$$

$$= \frac{1}{\left(1+e^{-z}\right)} - \left(\frac{1}{1+e^{-z}}\right)^2$$

$$= \phi(z) - \left(\phi(z)\right)^2$$

$$= \phi(z)\left(1 - \phi(z)\right)$$

$$= a(1-a)$$

Per comprendere meglio come calcoliamo il termine $\delta^{(2)}$, vediamo l'operazione più in dettaglio. Nell'equazione precedente, abbiamo moltiplicato la trasposta

$$\left(\boldsymbol{W}^{(2)}\right)^T$$

della matrice $t \times h$-dimensionale $\boldsymbol{W}^{(2)}$; *t* è il numero delle etichette delle classi di output e *h* è il numero delle unità nascoste. Ora

$$\left(\boldsymbol{W}^{(2)}\right)^T$$

diviene una matrice $h \times t$-dimensionale con $\delta^{(2)}$, che è un vettore $t \times 1$-dimensionale. Poi abbiamo eseguito una moltiplicazione a coppie fra

$$\left(\boldsymbol{W}^{(2)}\right)^T \delta^{(3)}$$

e

$$\left(a^{(2)} * \left(1 - a^{(2)}\right)\right)$$

che è anch'esso un vettore $t \times 1$-dimensionale. Alla fine, dopo aver ottenuto i δ termini, possiamo scrivere la derivata della funzione di costo nel seguente modo:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J\left(\boldsymbol{W}\right) = a_j^{(l)} \delta_i^{(l+1)}$$

Poi dobbiamo sommare la derivata parziale di ogni nodo *j*-esimo del livello *l* e l'errore *i*-esimo del nodo nel livello *l*+1:

$$\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

Ricordate che dobbiamo calcolare $\Delta_{i,j}^{(l)}$ per ogni campione del set di addestramento. Pertanto, è più facile implementarlo come una versione vettorializzata, come nella nostra precedente implementazione del MLP:

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} \left(\boldsymbol{A}^{(l)}\right)^T$$

Dopo aver sommato le derivate parziali, possiamo aggiungere il termine di regolarizzazione nel seguente modo:

$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} \text{ (tranne per il termine di bias)}$$

Infine, dopo aver calcolato i gradienti, possiamo aggiornare i pesi prendendo un passo in direzione opposta rispetto al gradiente:

$$\boldsymbol{W}^{(l)} := \boldsymbol{W}^{(l)} - \eta\Delta^{(l)}$$

Per riepilogare, la Figura 12.12 mostra il funzionamento della retropropagazione.

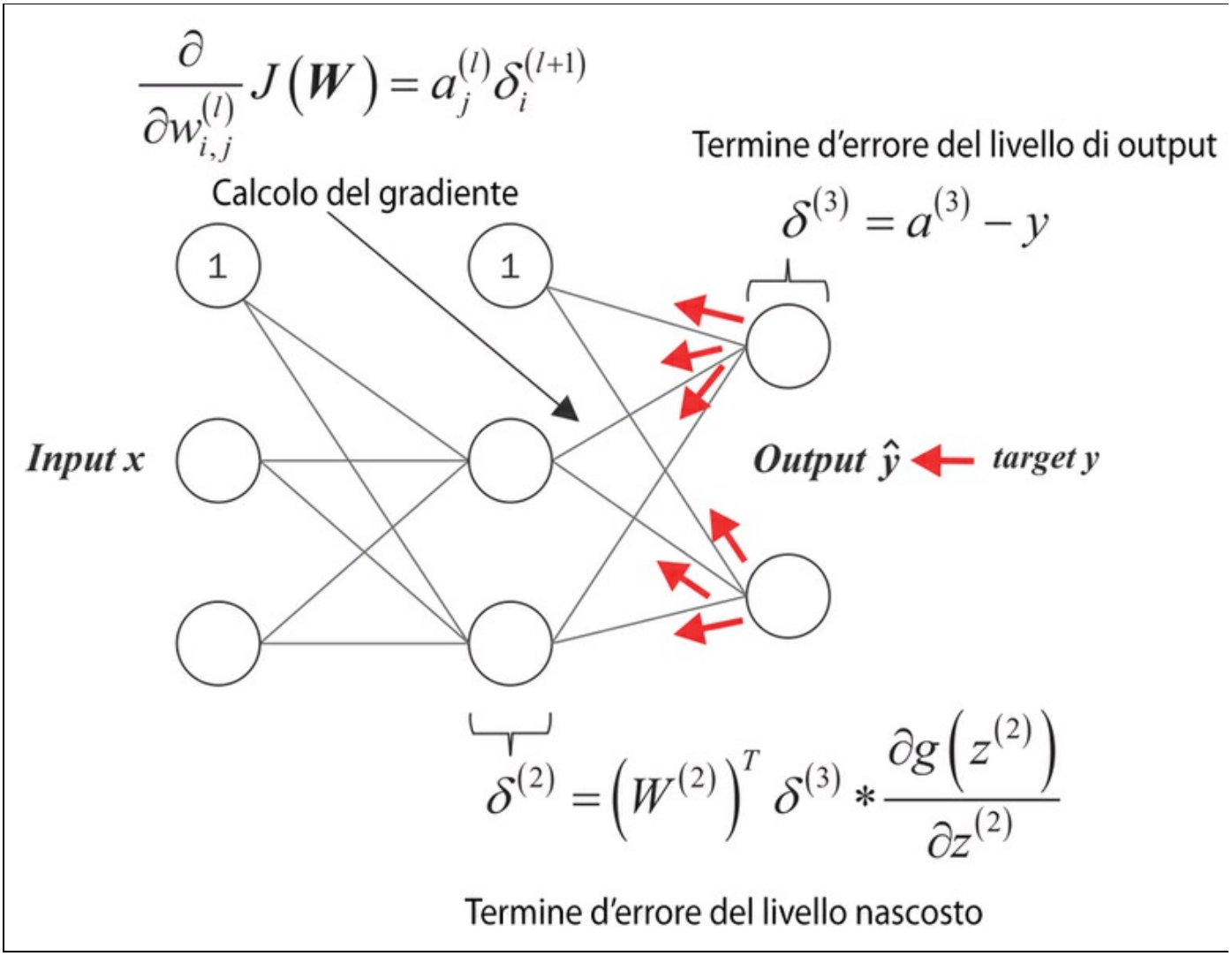


Figura 12.12

Aspetti intuitivi della retropropagazione

Sebbene la retroprogrammazione sia stata riscoperta e resa popolare circa trent'anni fa, rimane uno degli algoritmi più ampiamente utilizzati per addestrare in modo efficiente le reti neurali artificiali. In questo paragrafo, proponiamo un riepilogo più intuitivo del funzionamento di questo algoritmo così affascinante.

In pratica, la retropropagazione è solo un approccio computazionalmente molto efficiente per calcolare le derivate di una funzione di costo complessa. Il nostro obiettivo è quello di utilizzare queste derivate per conoscere i coefficienti di peso per i parametri da usare in una rete neurale artificiale multilivello. Il problema, nella parametrizzazione delle reti neurali, consiste nel fatto che in genere abbiamo a che fare con un numero molto esteso di coefficienti di peso in uno spazio di caratteristiche a elevata dimensionalità. Rispetto ad altre funzioni di costo che abbiamo visto nei capitoli precedenti, la superficie d'errore della funzione di costo di una rete neurale non è convessa o arrotondata. In questa superficie dei costi a elevata dimensionalità troviamo molte "buche" (ovvero minimi locali) che dobbiamo risolvere in modo da trovare il minimo globale della funzione di costo.

Potreste ricordarvi il concetto della regola di concatenamento delle classi di algebra. La regola di concatenamento è un approccio per la derivazione di una funzione complessa, nidificata, per esempio

$$f\left(g\left(x\right)\right)=y$$

che viene suddivisa nei suoi componenti di base:

$$\frac{\partial y}{\partial x}=\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

Nel contesto dell'algebra per computer, è stato sviluppato un insieme di tecniche per risolvere in modo molto efficiente questi problemi, grazie alla *differenziazione automatica*. Se siete interessati all'uso della differenziazione automatica in applicazioni di machine learning, vi consiglio di far riferimento alla seguente risorsa: A. G. Baydin e B. A. Pearlmutter, *Automatic Differentiation of Algorithms for Machine Learning*, arXiv preprint arXiv:1404.7456, 2014, liberamente disponibile su arXiv l'indirizzo http://arxiv.org/pdf/1404.7456.pdf.

La differenziazione automatica prevede due modalità: *diretta* e *inversa*. La retropropagazione è semplicemente un caso speciale della differenziazione automatica a modalità inversa. Il punto chiave è che l'applicazione della regola di concatenamento nella modalità diretta può essere piuttosto costosa, in quanto dovremmo moltiplicare grosse matrici per ciascun livello che alla fine moltiplichiamo per un vettore per ottenere l'output. Il trucco della modalità inversa è il fatto che partiamo da destra e procediamo verso sinistra: moltiplichiamo una matrice per un vettore, che ci fornisce un altro vettore che viene moltiplicato per la successiva matrice e così via. La moltiplicazione fra matrice e vettore è molto più economica dal punto di vista computazionale rispetto a una moltiplicazione fra matrici, motivo per cui la retropropagazione è uno degli algoritmi più utilizzati nell'addestramento delle reti neurali.

Debugging delle reti neurali con il controllo dei gradienti

Le implementazioni delle reti neurali artificiali possono essere piuttosto complesse ed è sempre opportuno controllare *manualmente* di aver implementato correttamente la retropropagazione. In questo paragrafo parleremo di una semplice procedura chiamata *controllo dei gradienti*, che è sostanzialmente un confronto fra i nostri gradienti analitici nella rete e i gradienti numerici. Il controllo dei gradienti non è specifico delle reti neurali ad avanzamento, ma può essere applicato a ogni altra architettura a rete neurale che utilizzi l'ottimizzazione basata su gradienti. Anche se prevedete di implementare algoritmi più semplici utilizzando l'ottimizzazione basata su gradienti, come la regressione lineare, la regressione logistica e le macchine a vettori di supporto, generalmente non sarebbe male controllare se i gradienti sono calcolati correttamente.

Nei paragrafi precedenti, abbiamo definito una funzione di costo $J(\boldsymbol{W})$ dove $\boldsymbol{W}$ è la matrice dei coefficienti di peso di una rete neurale. Notate che $J(\boldsymbol{W})$ è (sostanzialmente) una matrice costituita dalle matrici $\boldsymbol{W}^{(1)}$ e $\boldsymbol{W}^{(2)}$ in un perceptron multilivello con una unità nascosta. Abbiamo definito $\boldsymbol{W}^{(1)}$ come la matrice $h \times [m+1]$-dimensionale che connette il livello di input con il livello nascosto, dove *h* è il numero di unità nascoste e *m* è il numero di caratteristiche (unità di input). La matrice $\boldsymbol{W}^{(2)}$ che connette il livello nascosto con il livello di output ha dimensioni $t \times h$, dove *t* è il numero di unità di output. Abbiamo poi calcolato la derivata della funzione di costo per un peso $w_{i,j}^{(l)}$ nel seguente modo:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(\boldsymbol{W})$$

Ricordate che stiamo aggiornando i pesi compiendo un passo nella direzione opposta rispetto al gradiente. Nel controllo dei gradienti, confrontiamo questa soluzione analitica con un gradiente approssimato numericamente:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(\boldsymbol{W}) \approx \frac{J\left(w_{i,j}^{(l)} + \varepsilon\right) - J\left(w_{i,j}^{(l)}\right)}{\varepsilon}$$

Qui, ε è tipicamente un numero molto piccolo, per esempio 1e-5 (notate che 1e-5 è semplicemente una notazione più comoda del numero 0.00001). Intuitivamente, possiamo considerare questa differenza finita di approssimazione come la pendenza della linea secante che connette i due punti della funzione di costo per i due pesi w e $w + \varepsilon$ (entrambi valori scalari) come illustrato nella Figura 12.13. Per semplicità abbiamo omesso gli indici.

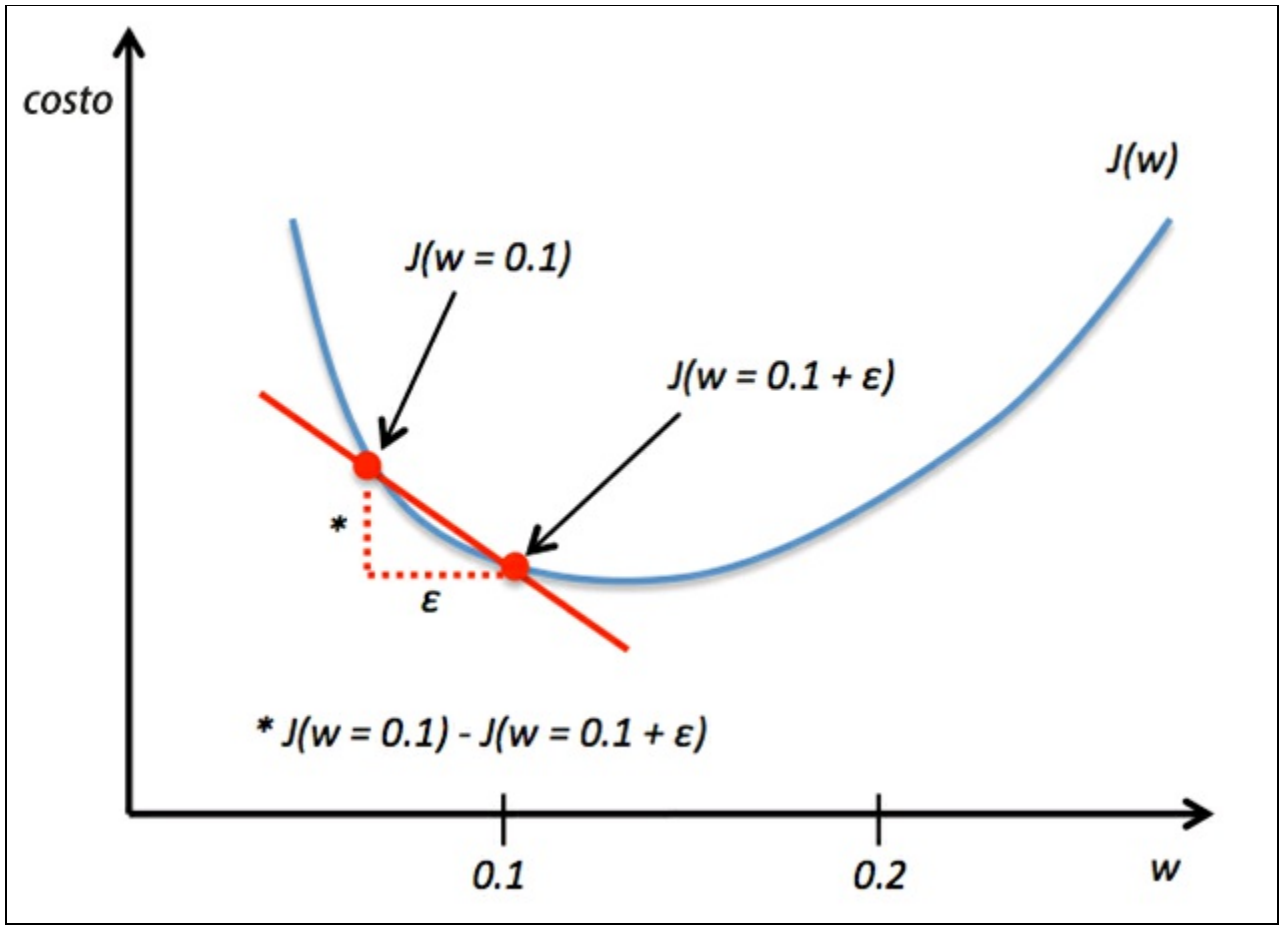


Figura 12.13

Un approccio ancora migliore, nel senso che fornisce un'approssimazione più precisa del gradiente, consiste nel calcolare il quoziente di differenza simmetrica (o centrata) fornito dalla seguente formula:

$$\frac{J\left(w_{i,j}^{(l)}+\varepsilon\right)-J\left(w_{i,j}^{(l)}-\varepsilon\right)}{2\varepsilon}$$

Tipicamente, la differenza approssimata fra il gradiente numerico J'_n e il gradiente analitico J'_a viene poi calcolata come la norma del vettore L2. Per motivi pratici, trasformiamo le matrici dei gradienti calcolate in vettori piatti, in modo da poter calcolare l'errore (la differenza fra i vettori dei gradienti) in modo più comodo:

$$errore = \left\| J'_n - J'_a \right\|_2$$

Un problema è il fatto che l'errore non è invariante rispetto alla scala (piccoli errori sono più significativi se le normali del vettore peso sono anch'esse piccole). Pertanto, è consigliabile calcolare una differenza normalizzata:

$$errore\ relativo = \frac{\left\| J'_n - J'_a \right\|_2}{\left\| J'_n \right\|_2 + \left\| J'_a \right\|_2}$$

Ora, vogliamo che l'errore relativo fra il gradiente numerico e il gradiente analitico sia il più piccolo possibile. Prima di implementare il controllo dei gradienti, dobbiamo discutere di un altro dettaglio: cosa si può considerare accettabile, come soglia d'errore, per passare il controllo dei gradienti? La soglia d'errore relativo per passare il controllo dei gradienti dipende dalla complessità dell'architettura della rete. Approssimativamente, più livelli nascosti aggiungiamo, maggiore sarà la differenza fra il gradiente numerico e quello analitico, sempre che la retropropagazione sia implementata correttamente. Poiché abbiamo implementato un'architettura di rete neurale relativamente semplice, vogliamo essere piuttosto rigidi in termini di soglia e definire le seguenti regole.

- Un errore relativo <= 1e-7 significa che tutto va bene.
- Un errore relativo <= 1e-4 significa che la condizione è problematica e dovremmo controllarla.
- Un errore relativo > 1e-4 significa che probabilmente c'è qualche errore nel codice.

Ora che abbiamo stabilito queste regole di base, implementiamo il controllo dei gradienti. Per farlo, possiamo semplicemente prendere la classe NeuralNetMLP che abbiamo implementato precedentemente e aggiungere il seguente metodo al corpo della classe:

```
def _gradient_checking(self, X, y_enc, w1,
                       w2, epsilon, grad1, grad2):
    """ Apply gradient checking (for debugging only)
    Returns
    ---------
    relative_error : float
      Relative error between the numerically
      approximated gradients and the backpropagated gradients.
    """
    num_grad1 = np.zeros(np.shape(w1))
    epsilon_ary1 = np.zeros(np.shape(w1))
    for i in range(w1.shape[0]):
        for j in range(w1.shape[1]):
            epsilon_ary1[i, j] = epsilon
            a1, z2, a2, z3, a3 = self._feedforward(
                                    X,
                                    w1 - epsilon_ary1,
                                    w2)
            cost1 = self._get_cost(y_enc,
                                   a3,
                                   w1-epsilon_ary1,
                                   w2)
            a1, z2, a2, z3, a3 = self._feedforward(
                                   X,
                                   w1 + epsilon_ary1,
                                   w2)
            cost2 = self._get_cost(y_enc,
                                   a3,
                                   w1 + epsilon_ary1,
                                   w2)
            num_grad1[i, j] = (cost2 - cost1) / (2 * epsilon)
            epsilon_ary1[i, j] = 0
    num_grad2 = np.zeros(np.shape(w2))
    epsilon_ary2 = np.zeros(np.shape(w2))
    for i in range(w2.shape[0]):
        for j in range(w2.shape[1]):
            epsilon_ary2[i, j] = epsilon
            a1, z2, a2, z3, a3 = self._feedforward(
                                    X,
                                    w1,
                                    w2 - epsilon_ary2)
            cost1 = self._get_cost(y_enc,
                                   a3,
                                   w1,
                                   w2 - epsilon_ary2)
            a1, z2, a2, z3, a3 = self._feedforward(                    X,
                                   w1,
                                   w2 + epsilon_ary2)
            cost2 = self._get_cost(y_enc,
                                   a3,
                                   w1,
                                   w2 + epsilon_ary2)
            num_grad2[i, j] = (cost2 - cost1) / (2 * epsilon)
            epsilon_ary2[i, j] = 0
    num_grad = np.hstack((num_grad1.flatten(),
                          num_grad2.flatten()))
    grad = np.hstack((grad1.flatten(), grad2.flatten()))
    norm1 = np.linalg.norm(num_grad - grad)
    norm2 = np.linalg.norm(num_grad)
    norm3 = np.linalg.norm(grad)
    relative_error = norm1 / (norm2 + norm3)
    return relative_error
```

Il codice di _gradient_checking sembra piuttosto semplice. Tuttavia, il consiglio è quello di assicurarsi che sia sempre il più semplice possibile. Il nostro obiettivo è

controllare bene il calcolo del gradiente, per assicurarci di non aver introdotto ulteriori errori nel controllo dei gradienti, scrivendo codice efficiente ma complesso. Poi abbiamo solo bisogno di applicare leggere modifiche al metodo `fit`. Nel seguente codice, abbiamo omesso il codice iniziale della funzione `fit` per motivi di chiarezza; le uniche righe che dobbiamo aggiungere al metodo sono implementate fra i commenti `##start gradient checking` e `## end gradient checking`:

```
class MLPGradientCheck(object):
  [...]
  def fit(self, X, y, print_progress=False):
    [...]
          # compute gradient via backpropagation
          grad1, grad2 = self._get_gradient(
                              a1=a1,
                              a2=a2,
                              a3=a3,
                              z2=z2,
                              y_enc=y_enc[:, idx],
                              w1=self.w1,
                              w2=self.w2)

          ## start gradient checking
          grad_diff = self._gradient_checking(
                             X=X[idx],
                             y_enc=y_enc[:, idx],
                             w1=self.w1,
                             w2=self.w2,
                             epsilon=1e-5,
                             grad1=grad1,
                             grad2=grad2)
          if grad_diff <= 1e-7:
             print('Ok: %s' % grad_diff)
          elif grad_diff <= 1e-4:
             print('Warning: %s' % grad_diff)
          else:
             print('PROBLEM: %s' % grad_diff)

          ## end gradient checking

          # update weights; [alpha * delta_w_prev]
          # for momentum learning
          delta_w1 = self.eta * grad1
          delta_w2 = self.eta * grad2
          self.w1 -= (delta_w1 +\
                  (self.alpha * delta_w1_prev))
          self.w2 -= (delta_w2 +\
                  (self.alpha * delta_w2_prev))
          delta_w1_prev = delta_w1
          delta_w2_prev = delta_w2
    return self
```

Supponendo di aver chiamato la classe perceptron multilivello modificata con il nome `MLPGradientCheck`, possiamo ora inizializzare un nuovo MLP con dieci livelli nascosti. Inoltre, disabilitiamo la regolarizzazione, l'apprendimento adattativo e il *momentum learning*. Quindi utilizziamo la discesa del gradiente standard, impostando `minibatches` uguale a 1. Il codice è il seguente:

```
>>> nn_check = MLPGradientCheck(n_output=10,
                        n_features=X_train.shape[1],
                        n_hidden=10,
                        l2=0.0,
                        l1=0.0,
                        epochs=10,
                        eta=0.001,
                        alpha=0.0,
```

```
                  decrease_const=0.0,
                  minibatches=1,
                  random_state=1)
```

Un difetto del controllo dei gradienti è il fatto che è molto, molto costoso dal punto di vista computazionale. L'addestramento di una rete neurale con il controllo dei gradienti attivato è talmente lenta che è opportuno effettuarla solo per scopi di debugging. Per questo motivo, è pratica comune eseguire il controllo dei gradienti solo su una manciata di campioni di addestramento (in questo caso abbiamo scelto 5). Il codice è il seguente:

```
>>> nn_check.fit(X_train[:5], y_train[:5], print_progress=False)
Ok: 2.56712936241e-10
Ok: 2.94603251069e-10
Ok: 2.37615620231e-10
Ok: 2.43469423226e-10
Ok: 3.37872073158e-10
Ok: 3.63466384861e-10
Ok: 2.22472120785e-10
Ok: 2.33163708438e-10
Ok: 3.44653686551e-10
Ok: 2.17161707211e-10
```

Come possiamo vedere dall'output, il nostro perceptron multilivello passa questo test con risultati eccellenti.

chiedervi perché non applicare l'apprendimento mini-batch per addestrare la nostra rete neurale per la classificazione delle cifre scritte a mano. Potreste ricordare la nostra discussione sulla discesa del gradiente stocastica che abbiamo utilizzato per implementare l'apprendimento online. Nell'apprendimento online, calcoliamo il gradiente sulla base di un unico esempio di addestramento ($k = 1$) alla volta per eseguire l'aggiornamento dei pesi. Sebbene questo sia un approccio stocastico, spesso conduce a soluzioni molto imprecise, con una convergenza molto più rapida rispetto alla discesa del gradiente standard. L'apprendimento mini-batch è una forma speciale di discesa del gradiente stocastica, dove calcoliamo il gradiente sulla base di un sottoinsieme *k* degli *n* campioni di addestramento, con $1 < k < n$. L'apprendimento mini-batch presenta il vantaggio rispetto l'apprendimento online che possiamo utilizzare la nostra implementazione vettorializzata per migliorarne l'efficienza computazionale. Tuttavia, possiamo aggiornare i pesi più velocemente rispetto alla normale discesa del gradiente. Intuitivamente, si può considerare l'apprendimento mini-batch come la previsione del risultato di un'elezione presidenziale partendo da un campione rappresentativo della popolazione, senza domandare all'intera popolazione.

Inoltre, abbiamo aggiunto ulteriori parametri di ottimizzazione, come la costante di crescita e un parametro per un tasso di apprendimento adattativo. Il motivo è che le reti neurali sono molto più difficili da addestrare rispetto ad algoritmi più semplici come Adaline, la regressione logistica o la macchina a vettori di supporto. Nelle reti neurali multilivello, in genere abbiamo centinaia, migliaia o anche miliardi di pesi da ottimizzare. Sfortunatamente, la funzione di output ha una superficie "frastagliata" e l'algoritmo di ottimizzazione può facilmente rimanere intrappolato all'interno di minimi locali, come mostra la Figura 12.14.

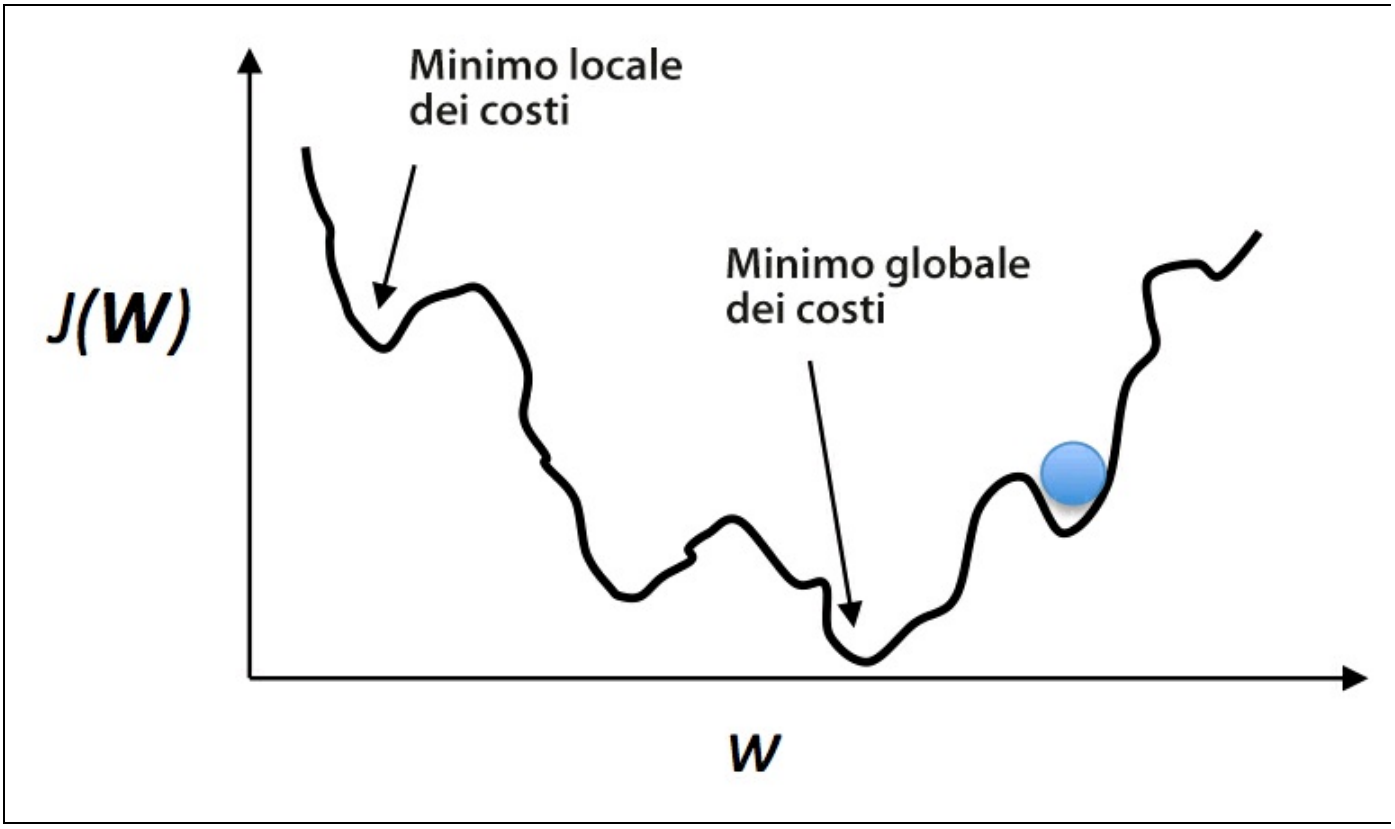


Figura 12.14

Notate che questa rappresentazione è estremamente semplificata, in quanto la nostra rete neurale presenta molte dimensioni; è praticamente impossibile rappresentare graficamente l'effettiva superficie dei costi. Qui mostriamo solo la superficie dei costi per un unico peso sull'asse *x*. Tuttavia, è importante che il nostro algoritmo non rimanga intrappolato in un minimo locale. Incrementando il tasso di apprendimento, possiamo riuscire a sfuggire con maggiore facilità da questo tipo di minimi locali. D'altra parte, incrementiamo anche la possibilità di non riuscire a centrare bene l'ottimo globale se il tasso d'apprendimento risulta troppo ampio. Poiché inizializziamo i pesi in modo casuale, iniziamo con una soluzione al problema di ottimizzazione che tipicamente è del tutto sbagliata. Una costante di decrescita, che abbiamo definito in precedenza, può aiutarci a discendere la superficie dei costi più velocemente all'inizio; il tasso di apprendimento adattativo ci consente di giungere meglio al minimo globale.

Altre architetture di reti neurali

In questo capitolo abbiamo trattato una delle più note rappresentazioni delle reti neurali ad avanzamento, il perceptron multilivello. Le reti neurali sono attualmente uno degli argomenti di ricerca più interessanti nel campo del machine learning, ma vi sono molti altri esempi di architetture di reti neurali che non rientrano negli scopi di questo libro. Se siete interessati aconoscere altre reti neurali e i relativi algoritmi di deep learning, potete leggere Y. Bengio, *Learning Deep Architectures for AI,* in "Foundations and Trends in Machine Learning", 2(1):1–127, 2009. Il libro è attualmente disponibile gratuitamente su http://www.iro.umontreal.ca/~bengioy/papers/ftml_book.pdf.

Le reti neurali sarebbero in realtà un argomento cui dedicare un intero libro, ma diamo almeno una breve occhiata ad altre due architetture molto note: le *reti neurali convolutive* e le *reti neurali ricorrenti*.

Reti neurali convolutive

Le *reti neurali convolutive* (*Convolutional Neural Networks, CNN* o *ConvNets*) hanno acquisito popolarità nelle applicazioni di visione computerizzata grazie alle loro eccezionali prestazioni nel campo della classificazione delle immagini. Al momento attuale, le CNN sono una delle architetture di rete neurale più utilizzate nel deep learning. L'idea base delle reti neurali convolutive si basa sul fatto di costruire più livelli di *rilevatori di caratteristiche,* per considerare la disposizione spaziale dei pixel in un'immagine di input. Notate che esistono molte varianti delle reti CNN. In questo paragrafo tratteremo solo i concetti generali di questa architettura. A chi fosse interessato all'argomento, è consigliata la lettura delle pubblicazioni di Yann LeCun (http://yann.lecun.com) che è uno dei co-inventori delle reti CNN. In particolare, sono consigliati i seguenti testi.

- Y. LeCun, L. Bottou, Y. Bengio e P. Haffner, *Gradient-based Learning Applied to Document Recognition*, in "Proceedings of the IEEE", 86(11):2278–2324, 1998.
- P. Y. Simard, D. Steinkraus e J. C. Platt, *Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis*, in "IEEE", 2003, p.958.

Come ricorderete dalla nostra implementazione del perceptron multilivello, abbiamo "disteso" le immagini in vettori delle caratteristiche e questi input sono stati completamente connessi al livello nascosto: l'informazione spaziale non è stata codificata in questa architettura di rete. Nelle reti CNN utilizziamo dei *campi recettori* per connettere il livello di input alla mappa delle caratteristiche. Questi campi recettori possono essere considerati come finestre sovrapposte che possiamo far scorrere sui pixel di un'immagine di input, in modo da creare una mappa delle caratteristiche. Le lunghezza della finestra a scorrimento così come le dimensioni della finestra sono ulteriori iperparametri del modello, che quindi dobbiamo definire a priori. Il processo di creazione della mappa delle caratteristiche è chiamato anche *convoluzione*. Un esempio di questo livello convolutivo, il livello che connette i pixel di input con ogni unità della mappa delle caratteristiche, è rappresentato nella Figura 12.15.

È importante notare che i rilevatori di caratteristiche vengono replicati, ovvero che i campi recettori che mappano le caratteristiche sulle unità del livello successivo condividono gli stessi pesi. Inoltre, l'idea chiave è che se il rilevatore di caratteristiche è utile in una parte dell'immagine, potrebbe essere utile anche in un'altra parte. Un ottimo effetto collaterale di questo approccio è che riduce notevolmente il numero di parametri che devono essere appresi. Poiché consentiamo che aree diverse dell'immagine possano essere rappresentate in modi diversi, le reti CNN sono particolarmente efficaci nel riconoscere gli oggetti di dimensioni differenti e con posizioni differenti all'interno di un'immagine. Non dobbiamo preoccuparci troppo dei problemi di cambio di scala e di centratura dell'immagine come avviene invece in MNIST.

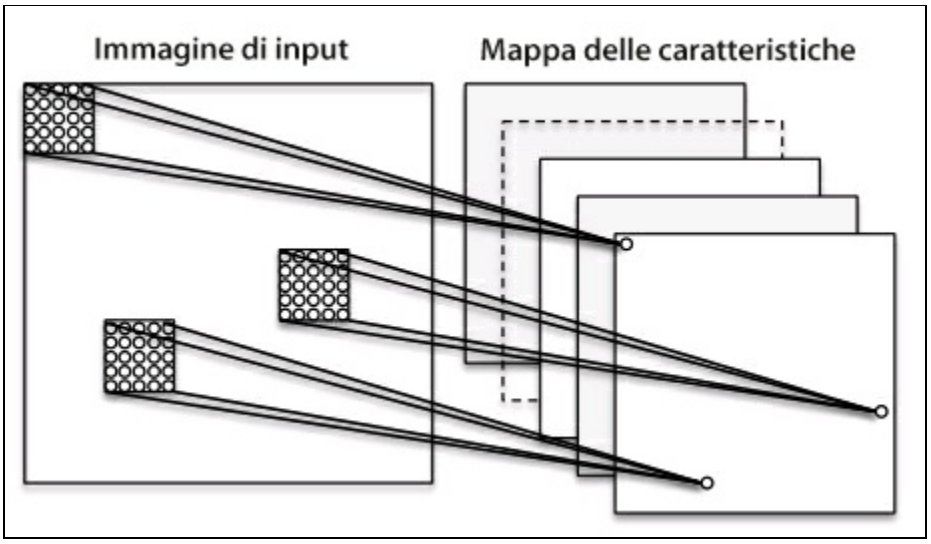


Figura 12.15

Nelle reti CNN, un livello convolutivo è seguito da un *livello pooling* (o di *sub-campionamento*). Nel pooling, riepiloghiamo i rilevatori di caratteristiche confinanti, con lo scopo di ridurre il numero di caratteristiche per il livello successivo. La tecnica di pooling può essere considerata come un semplice metodo di estrazione delle caratteristiche, dove prendiamo la media del valore massimo di una serie di caratteristiche ravvicinate e la passiamo al livello successivo. Per creare una rete neurale convolutiva profonda, impiliamo più livelli (alternati fra convolutivi e di pooling) prima di connetterli con un perceptron multilivello per la classificazione. Il tutto è rappresentato nella Figura 12.16.

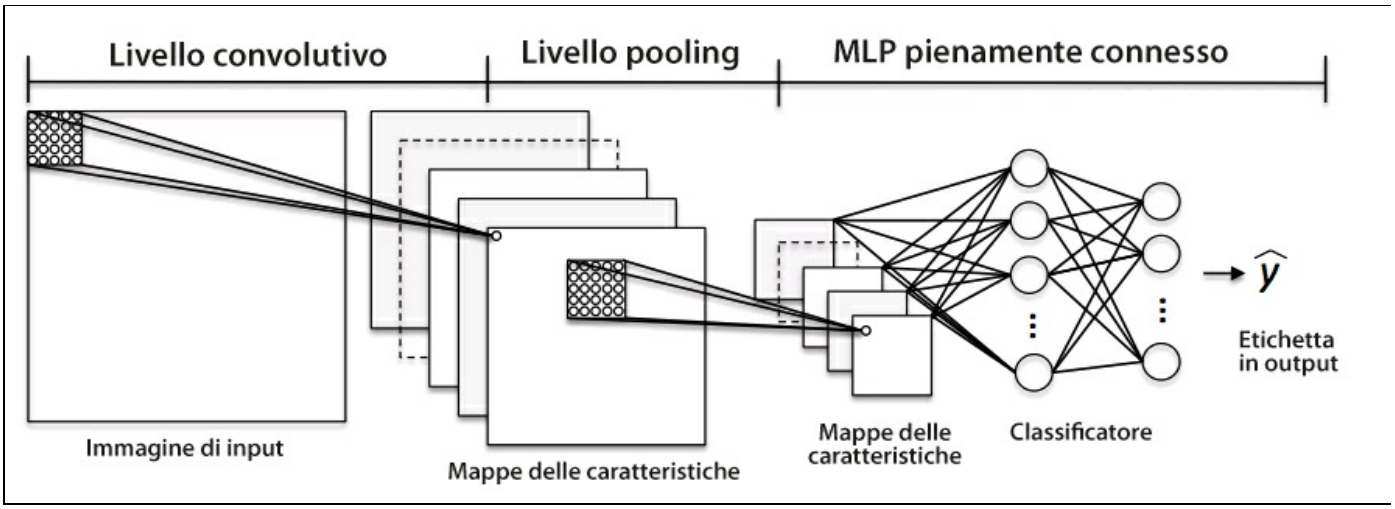


Figura 12.16

Reti neurali ricorrenti

Le reti neurali ricorrenti (*RNN – Recurrent Neural Networks*), possono essere considerate come reti neurali ad avanzamento, con cicli di feedback o momenti di retropropagazione. Nelle reti RNN, i neuroni operano solo per un tempo limitato, prima di essere (temporaneamente) disattivati. A loro volta, questi neuroni attivano altri neuroni che si accendono in momenti successivi. Sostanzialmente possiamo considerare le reti neurali ricorrenti come se fossero MLP con un tempo variabile aggiuntivo. Il componente temporale e la struttura dinamica consentono alla rete di utilizzare non solo gli input correnti, ma anche gli input incontrati in precedenza (Figura 12.17).

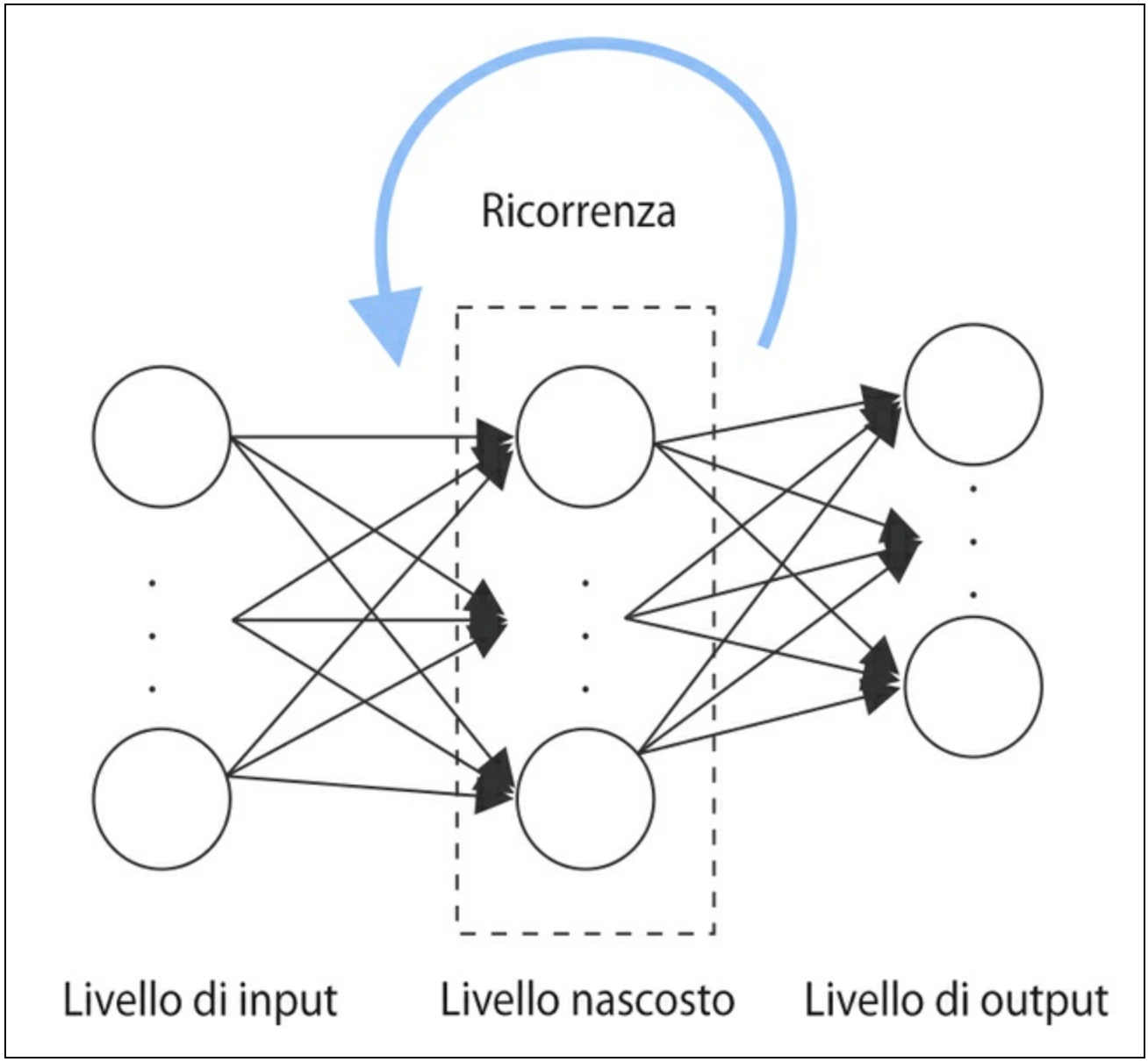


Figura 12.17

Sebbene le reti RNN offrano risultati notevoli nel riconoscimento del parlato, nella traduzione del linguaggio naturale e nel riconoscimento della scrittura manuale, si tratta di architetture di rete molto difficili da addestrare. Questo perché non possiamo semplicemente retropropagare l'errore livello per livello; dobbiamo considerare la componente temporale aggiuntiva, che amplifica i problemi della sparizione e dell'esplosione del gradiente. Nel 1997, proprio per risolvere questo problema, Juergen Schmidhuber e i suoi collaboratori hanno introdotto le cosiddette unità di memoria a lungo e breve termine (*LSTM – Long Short Term Memory*); S. Hochreiter e J. Schmidhuber, *Long Short-term Memory*, in "Neural Computation", 9(8):1735–1780, 1997.

Tuttavia, occorre notare che vi sono molte varianti delle reti RNN e una loro discussione dettagliata non rientra negli scopi di questo libro.

Un'ultima parola sull'implementazione delle reti neurali

Potreste chiedervi perché abbiamo affrontato tutti questi aspetti teorici solo per implementare una semplice rete artificiale multilivello in grado di classificare le cifre scritte a mano, invece di utilizzare una libreria di apprendimento open source di Python. Un motivo è che al momento attuale, scikit-learn non offre un'implementazione di un MLP. Ma soprattutto, noi che utilizziamo algoritmi di machine learning dobbiamo avere almeno una conoscenza minimale degli algoritmi che utilizziamo, così da poter applicare in modo appropriato e con successo le tecniche corrette di machine learning.

Ora che conosciamo il funzionamento delle reti neurali ad avanzamento, siamo pronti per esplorare le più sofisticate librerie Python basate su NumPy, come Theano (http://deeplearning.net/software/theano/), che ci consente di costruire delle reti neurali in modo molto efficiente. Esamineremo l'argomento nel Capitolo 13, *Parallelizzare l'addestramento delle reti neurali con Theano*. Nel corso degli ultimi due anni, Theano ha acquisito molta popolarità fra i ricercatori nel campo del machine learning, che lo utilizzano per realizzare reti neurali profonde grazie alla sua capacità di ottimizzare le espressioni matematiche per calcoli su array multidimensionali utilizzando le unità di elaborazione grafica (*GPU – Graphical Processing Units*).

Un'ottima raccolta di guide sull'uso di Theano si trova all'indirizzo http://deeplearning.net/software/theano/tutorial/index.html#tutorial.

Sono inoltre disponibili molte interessanti librerie, sviluppate attivamente, per addestrare le reti neurali in Theano, che vale la pena di monitorare:

- *Pylearn2* (http://deeplearning.net/software/pylearn2/);
- *Lasagne* (https://lasagne.readthedocs.org/en/latest/);
- *Keras* (http://keras.io).

Riepilogo

In questo capitolo, abbiamo trattato i concetti più importanti su cui si basano le reti neurali artificiali multilivello, che sono attualmente fra gli argomenti più interessanti nella ricerca che opera nell'ambito del machine learning. Nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*, abbiamo iniziato il nostro viaggio con semplici reti neurali monolivello e ora abbiamo connesso più neuroni per formare una potente architettura a rete neurale in grado di risolvere problemi complessi come il riconoscimento della scrittura manuale. Abbiamo esaminato il noto algoritmo di retropropagazione, che è uno degli elementi costitutivi di molti modelli a rete neurale che sono utilizzati nel deep learning. Dopo aver imparato il funzionamento dell'algoritmo di retropropagazione, siamo stati in grado di aggiornare i pesi di una complessa rete neurale. Abbiamo inoltre aggiunto utili modifiche, come l'apprendimento mini-batch e un tasso di apprendimento adattativo, che ci ha consentito di addestrare in modo più efficiente una rete neurale.

Parallelizzare l'addestramento delle reti neurali con Theano

Nel capitolo precedente, abbiamo trattato molti concetti matematici utili per comprendere il funzionamento delle reti neurali artificiali ad avanzamento e, in particolare, dei perceptron multilivello. Una buona conoscenza delle basi matematiche degli algoritmi di machine learning è molto importante, in quanto aiuta a utilizzare questi potenti algoritmi nel modo più efficace e *corretto*. Nel corso dei capitoli precedenti, avete dedicato molto tempo a imparare le migliori tecniche di machine learning e avete anche provato a implementare degli algoritmi partendo da zero. In questo capitolo le cose si faranno più distese e potrete, per così dire, riposarvi sugli allori: godetevi questo interessante viaggio attraverso una delle più potenti librerie utilizzate dai ricercatori nel campo del machine learning per sperimentare con le reti neurali profonde e addestrarle in modo molto efficace. La maggior parte della ricerca nel campo del machine learning utilizza computer dotati di potenti *GPU* (*Graphics Processing Units*). Se siete interessati ad approfondire l'argomento GPU, che è attualmente argomento più in voga nella ricerca dedicata al machine learning, questo capitolo è assolutamente imperdibile. Tuttavia, non preoccupatevi se il vostro computer non è dotato di una GPU; in questo capitolo, l'uso della GPU è del tutto opzionale.

Prima di iniziare, ecco una breve panoramica degli argomenti che toccheremo nel corso di questo capitolo.

- Scrivere codice di machine learning ottimizzato con Theano.
- Scegliere funzioni di attivazione per reti neurali artificiali.
- Utilizzare la libreria di deep learning Keras per sperimentazioni rapide e semplici.

Realizzare, compilare ed eseguire espressioni con Theano

In questo paragrafo, esploreremo uno strumento davvero potente: la libreria Theano, che è stata concepita per addestrare i modelli di machine learning in modo particolarmente efficace utilizzando Python. Lo sviluppo di Theano è iniziato nel 2008 presso i laboratori *LISA* (*Laboratoire d'Informatique des Systèmes Adaptatifs*, http://lisa.iro.umontreal.ca) sotto la guida di Yoshua Bengio.

Prima di parlare di Theano e di quello che può fare per noi per accelerare ogni compito di machine learning, parliamo di alcuni dei problemi che dobbiamo affrontare quando svolgiamo calcoli "costosi" sul nostro hardware. Fortunatamente, le prestazioni dei processori dei computer migliorano costantemente nel corso degli anni, e ciò ci consente di addestrare sistemi di apprendimento sempre più potenti e complessi, per migliorare le prestazioni predittive dei nostri modelli di machine learning. Anche il più economico computer desktop disponibile al giorno d'oggi è dotato di CPU multi-core. Nei capitoli precedenti, abbiamo visto che molte funzioni di scikit-learn ci consentono di distribuire i calcoli sulle varie unità di elaborazione. Tuttavia, per default, Python può operare su un solo core, a causa del *GIL* (*Global Interpreter Lock*). Tuttavia, sebbene possiamo sfruttare la sua libreria `multiprocessing` per distribuire i calcoli su più core, dobbiamo considerare che anche l'hardware desktop più avanzato solo raramente può contare su più di 8 o 16 core.

Se torniamo al capitolo precedente, nel quale abbiamo implementato un semplice perceptron multilivello con un unico livello nascosto costituito da 50 unità, abbiamo già dovuto ottimizzare circa 1000 pesi per delineare un modello rivolto a un semplicissimo compito di classificazione delle immagini. Le immagini contenute nel MNIST sono davvero piccole (28 × 28) e possiamo quindi immaginare l'esplosione del numero dei parametri qualora volessimo introdurre ulteriori livelli nascosti oppure lavorare con immagini che vantano densità di pixel superiori. Tale compito diverrebbe ben presto improbo per un'unica unità di elaborazione. Ora la domanda è come affrontare questi problemi in modo più efficace? La soluzione ovvia a questo problema consiste nell'utilizzo delle GPU. Le GPU sono veri cavalli da soma. In realtà, una scheda grafica è come un computer compatto, collocato all'interno del PC. Un altro vantaggio è il fatto che le GPU moderne sono

relativamente economiche rispetto alle migliori CPU, come possiamo vedere nella Tabella 13.1.

Tabella 13.1

Specifiche	Intel Core i7-5960X Extreme Edition	NVIDIA GeForce GTX 980 Ti
Frequenza di clock	3.0 GHz	1.0 GHz
Core	8	2816
Ampiezza di banda memoria	68 GB/s	336.5 GB/s
Calcoli in virgola mobile	354 GFLPS	5632 GFLPS
Prezzo	1000 euro	700 euro

Fra le fonti citabili a supporto di questi dati (alla data del 20 agosto 2015) trovate i seguenti siti web.

- http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980-ti/specifications.
- http://ark.intel.com/products/82930/Intel-Core-i7-5960XProcessor-Extreme-Edition-20M-Cache-up-to-3_50-GHz.

Al 70% del prezzo di una moderna CPU, possiamo ottenere una GPU che vanta un numero di core 450 volte superiore ed è in grado di svolgere calcoli in virgola mobile a una velocità 15 volte superiore. Pertanto, che cosa ci impedisce di utilizzare le GPU per le nostre attività di machine learning? Il compito di scrivere codice rivolto alle GPU non è però banale quanto quello di realizzare codice Python per il nostro interprete. Esistono speciali pacchetti, come CUDA e OpenCL, che consentono di utilizzare le GPU. Tuttavia, scrivere codice in CUDA o OpenCL non è esattamente come operare in un ambiente comodo per l'implementazione e l'esecuzione di algoritmi di machine learning. Esattamente questo è il motivo per cui è stato sviluppato Theano.

Che cos'è Theano

Che cos'è, esattamente, Theano? Un linguaggio di programmazione, un compilatore o una libreria Python? In realtà, è tutte queste cose: Theano è stato sviluppato per implementare, compilare e valutare espressioni matematiche in modo molto efficiente con particolare attenzione agli array multidimensionali (tensori). Offre un'opzione che consente di eseguire il codice sulle CPU, tuttavia la sua vera potenza si ottiene con l'impiego delle GPU per sfruttare la loro grande ampiezza di banda nell'uso della memoria e la loro grande capacità di eseguire calcoli in virgola mobile. Utilizzando Theano, possiamo eseguire con facilità codice in parallelo

anche nella memoria condivisa. Nel 2010, gli sviluppatori di Theano rilevavano prestazioni 1.8 volte più veloci rispetto a NumPy quando il codice veniva eseguito sulla sola CPU, mentre quando Theano poteva utilizzare la GPU, risultava ben 11 volte più veloce di NumPy (J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley e Y. Bengio, *Theano: A CPU and GPU Math Compiler in Python*, in "Proceedings 9th Python in Science Conference", pp. 1–7, 2010). Tenete in considerazione che questo benchmark risale al 2010 e che da allora Theano è migliorato sensibilmente, così come le capacità di elaborazione delle nuove schede grafiche.

Pertanto, quale relazione c'è fra Theano e NumPy? Theano si basa su NumPy e ha una sintassi molto simile, il che rende il suo uso molto comodo per coloro che conoscono già l'utilizzo di quest'ultimo. Occorre anche dire che Theano non è affatto, semplicemente, un "NumPy con gli steroidi", come molte persone amano descriverlo, ma condivide anche alcune analogie con SymPy (http://www.sympy.org), un pacchetto Python per il calcolo simbolico (per l'algebra simbolica). Come abbiamo visto nei capitoli precedenti, in NumPy descriviamo le nostre variabili e il modo in cui vogliamo combinarle; poi il codice viene eseguito riga per riga. In Theano, invece, scriviamo innanzitutto il problema e la descrizione del modo in cui vogliamo analizzarlo. Poi, Theano ottimizza e compila il codice per noi utilizzando C/C++ o CUDA/OpenCL se vogliamo utilizzare per l'esecuzione le GPU. Per generare il codice ottimizzato, Theano deve conoscere le caratteristiche del problema e ragionarci considerandolo un albero di operazioni (o un grafico di espressioni simboliche). Notate che Theano è ancora in fase di sviluppo e gli vengono aggiunte sempre nuove funzionalità e sempre nuovi affinamenti. In questo capitolo, esporremo i concetti di base di Theano e vedremo come utilizzarlo per compiti di machine learning. Poiché Theano è una libreria eccezionale, dotata di molte funzionalità avanzate, sarebbe impossibile descriverle tutte in questo libro. Tuttavia troverete utili link all'eccellente documentazione online (http://deeplearning.net/software/theano/), di grande importanza per chiunque volesse conoscere qualcosa di più su questa libreria.

Primi passi con Theano

A seconda della configurazione del vostro sistema, potete semplicemente installare Theano da PyPI lanciando il seguente comando dal Terminale:

```
pip install Theano
```

Se doveste avere problemi con la procedura di installazione, leggete le raccomandazioni sul sistema e la piattaforma riportate all'indirizzo http://deeplearning.net/software/theano/install.html. Notate che tutto il codice di questo capitolo può essere eseguito anche solo sulla CPU; l'uso della GPU è del tutto opzionale, ma consigliato per sfruttare appieno tutti i vantaggi di Theano. Se avete una scheda grafica che supporta CUDA o OpenCL, consultate la guida aggiornata su http://deeplearning.net/software/theano/tutorial/using_gpu.html#using-gpu, per configurare il sistema in modo appropriato.

Sostanzialmente, Theano si basa sui cosiddetti *tensori* per la valutazione di espressioni matematiche simboliche. I tensori possono essere considerati come una generalizzazione degli scalari, dei vettori, delle matrici e così via. In termini più concreti, uno scalare può essere definito come un tensore di rango 0, un vettore come un tensore di rango 1, una matrice come un tensore di rango 2 e le matrici tridimensionali come sensori di rango 3. Come esercizio di "riscaldamento", partiremo con l'uso di semplici scalari dal modulo `tensor` di Theano per calcolare un input *z* di un punto campione *x* in un dataset monodimensionale con peso w_1 e bias w_0:

$$z = x_1 \times w_1 + w_0$$

Il codice è il seguente:

```
>>> import theano
>>> from theano import tensor as T
# initialize
>>> x1 = T.scalar()
>>> w1 = T.scalar()
>>> w0 = T.scalar()
>>> z1 = w1 * x1 + w0
# compile
>>> net_input = theano.function(inputs=[w1, x1, w0],
...                             outputs=z1)
# execute
>>> print('Net input: %.2f' % net_input(2.0, 1.0, 0.5))
Net input: 2.50
```

Tutto molto semplice, vero? Nello scrivere codice Theano, dobbiamo seguire tre semplici passi: definire i *simboli* (gli oggetti `Variable`), compilare il codice e infine eseguirlo. Nel passo di inizializzazione, abbiamo definito tre simboli, `x1`, `w1` e `w0`, per calcolare `z1`. Poi abbiamo compilato una funzione `net_input` per calcolare l'input `z1`.

Tuttavia, vi è un dettaglio che merita particolare attenzione quando si scrive codice Theano: il tipo delle nostre variabili (`dtype`). Potete considerarlo una benedizione o un fardello, ma in Theano dovete scegliere se utilizzare numeri interi

o in virgola mobile a 64 o a 32 bit, il che influenza notevolmente le prestazioni del codice. Parleremo dei vari tipi di variabili più in dettaglio nel prossimo paragrafo.

Configurazione di Theano

Al giorno d'oggi, indipendentemente dal fatto che utilizzate Mac OS X, Linux o Microsoft Windows, impiegate software e applicazioni che utilizzano indirizzi di memoria a 64 bit. Tuttavia, per accelerare la valutazione delle espressioni matematiche sulle GPU, spesso si impiegano vecchi indirizzi di memoria a 32 bit. Attualmente, questa è l'unica architettura di calcolo supportata da Theano. In questo paragrafo vedremo come configurare in modo appropriato Theano. Se siete interessati ai dettagli sulla configurazione di Theano, potete consultare la documentazione online su http://deeplearning.net/software/theano/library/config.html.

Quando implementiamo degli algoritmi di machine learning, impieghiamo soprattutto numeri in virgola mobile. Per default, sia NumPy, sia Theano utilizzano il formato in virgola mobile a doppia precisione (`float64`). Tuttavia, sarebbe davvero utile impiegare indifferentemente `float64` (CPU) e `float32` (GPU) nello sviluppo di codice Theano per la prototipizzazione su CPU e l'esecuzione su GPU. Per esempio, per accedere alle impostazioni predefinite per le variabili in virgola mobile di Theano, potete eseguire il codice seguente nell'interprete Python:

```
>>> print(theano.config.floatX)
float64
```

Se non avete modificato alcuna impostazione dopo l'installazione di Theano, l'impostazione predefinita per i numeri in virgola mobile dovrebbe essere `float64`. Tuttavia, potete cambiarla in `float32` nella sessione corrente di Python utilizzando il codice seguente:

```
>>> theano.config.floatX = 'float32'
```

Notate che, sebbene attualmente l'utilizzo della GPU in Theano richieda tipi `float32`, potete utilizzare su CPU entrambi i tipi, `float64` e `float32`. Pertanto, per cambiare globalmente le impostazioni di default, potete modificare la variabile `THEANO_FLAGS` tramite il Terminale dalla riga di comando (Bash):

```
export THEANO_FLAGS=floatX=float32
```

Alternativamente, potete applicare queste impostazioni solo a un determinato script Python, lanciandolo nel seguente modo:

```
THEANO_FLAGS=floatX=float32 python your_script.py
```

Finora abbiamo visto come configurare i tipi in virgola mobile standard, in modo da sfruttare al meglio la GPU utilizzando Theano. Ora vedremo come scambiare l'esecuzione fra la CPU e la GPU. Se eseguiamo il codice seguente, possiamo controllare se stiamo utilizzando la CPU o la GPU:

```
>>> print(theano.config.device)
cpu
```

Il consiglio è quello di utilizzare come default `cpu`, per facilitare la prototipizzazione e il debugging del codice. Per esempio, potete lanciare il codice Theano sulla CPU eseguendolo in uno script, come nel seguente comando:

```
THEANO_FLAGS=device=cpu,floatX=float64 python your_script.py
```

Tuttavia, dopo aver implementato il codice, volendolo eseguire in modo più efficiente tramite l'hardware GPU, potete utilizzare il codice seguente senza ulteriori interventi sul codice originale:

```
THEANO_FLAGS=device=gpu,floatX=float32 python your_script.py
```

Può anche essere comodo creare un file `.theanorc` nella directory home per rendere permanenti queste configurazioni. Per esempio, per utilizzare sempre `float32` e GPU, potete creare un file `.theanorc` che includa queste impostazioni. Il comando è il seguente:

```
echo -e "\n[global]\nfloatX=float32\ndevice=gpu\n" >> ~/.theanorc
```

Se non utilizzate un Terminale di Mac OS X o Linux, potete creare manualmente un file `.theanorc` utilizzando un qualsiasi editor di testi, aggiungendogli le seguenti righe:

```
[global]
floatX=float32
device=gpu
```

Ora che sapete come configurare in modo appropriato Theano rispetto all'hardware disponibile, possiamo occuparci di strutture ad array complesse, argomento del prossimo paragrafo.

Utilizzo delle strutture ad array

In questo paragrafo vedremo come impiegare le strutture ad array in Theano utilizzando il suo modulo `tensor`. Tramite il codice seguente, creeremo una semplice matrice 2 × 3 e calcoleremo le somme delle colonne utilizzando le espressioni ottimizzate a tensori di Theano:

```
>>> import numpy as np
# initialize
# if you are running Theano on 64 bit mode,
# you need to use dmatrix instead of fmatrix
>>> x = T.fmatrix(name='x')
>>> x_sum = T.sum(x, axis=0)
```

```
# compile
>>> calc_sum = theano.function(inputs=[x], outputs=x_sum)
# execute (Python list)
>>> ary = [[1, 2, 3], [1, 2, 3]]
>>> print('Column sum:', calc_sum(ary))
Column sum: [ 2.  4.  6.]
# execute (NumPy array)
>>> ary = np.array([[1, 2, 3], [1, 2, 3]],
...                dtype=theano.config.floatX)
>>> print('Column sum:', calc_sum(ary))
Column sum: [ 2.  4.  6.]
```

Come abbiamo visto in precedenza, i tre semplici passi da usare per impiegare Theano sono: definire le variabili, compilare il codice ed eseguirlo. L'esempio precedente mostra che Theano è in grado lavorare anche coi tipi Python e NumPy: `list` e `numpy.ndarray`.

Approfondimento

Notate che abbiamo utilizzato l'argomento opzionale `name` (qui, `x`) quando abbiamo creato la `TensorVariable fmatrix`, il che può essere utile per il debugging del codice o la stampa del grafico di Theano. Per esempio, se avessimo stampato il simbolo `fmatrix x` senza fornirgli un nome `name`, la funzione `print` ci avrebbe restituito il suo `TensorType`:

```
>>> print(x)
<TensorType(float32, matrix)>
```

Al contrario, se `TensorVariable` viene inizializzata con un argomento `name x`, come nel nostro esempio precedente, questa verrà restituita dalla funzione `print`:

```
>>> print(x)
x
```

L'accesso al `TensorType` può essere ottenuto tramite il metodo `type`:

```
>>> print(x.type())
<TensorType(float32, matrix)>
```

Theano offre anche un sistema di gestione della memoria molto avanzato, che, per migliorare le prestazioni, riutilizza la memoria. Più concretamente, Theano distribuisce lo spazio di memoria su più dispositivi, CPU e GPU; per controllare le variazioni nello spazio di memoria, Theano duplica i rispettivi buffer. Poi, esamineremo la variabile `shared`, che ci consente di disperdere grossi oggetti (array) e offre varie funzioni di accesso in lettura e scrittura, in modo da poter svolgere aggiornamenti su questi oggetti dopo la compilazione. Purtroppo una descrizione dettagliata della gestione della memoria in Theano non rientra negli scopi di questo libro. Pertanto siete incoraggiati a consultare le informazioni aggiornate su Theano e la sua gestione della memoria all'indirizzo http://deeplearning.net/software/theano/tutorial/aliasing.html.

```
# initialize
>>> x = T.fmatrix('x')
>>> w = theano.shared(np.asarray([[0.0, 0.0, 0.0]],
                      dtype=theano.config.floatX))
>>> z = x.dot(w.T)
>>> update = [[w, w + 1.0]]
# compile
>>> net_input = theano.function(inputs=[x],
...                      updates=update,
```

```
...                         outputs=z)
# execute
>>> data = np.array([[1, 2, 3]],
...                 dtype=theano.config.floatX)
>>> for i in range(5):
...     print('z%d:' % i, net_input(data))
z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]
```

Come potete vedere, la condivisione della memoria tramite Theano è molto semplice: nell'esempio precedente, abbiamo definito una variabile update nella quale abbiamo dichiarato di voler aggiornare un array w con un valore 1.0 dopo ogni iterazione del ciclo for. Dopo aver definito quale oggetto volevamo aggiornare e in quale modo, abbiamo passato questa informazione al parametro update del compilatore theano.function.

Un altro trucco interessante di Theano è l'uso della variabile givens per inserire dei valori nel grafico prima della compilazione. Utilizzando questo approccio, possiamo ridurre il numero di trasferimenti dal programma alle CPU e alle GPU, per accelerare gli algoritmi di apprendimento che impiegano variabili condivise. Se utilizziamo il parametro inputs in theano.function, i dati vengono trasferiti dalla CPU alla GPU più volte, per esempio, se iteriamo più volte su un dataset (come nelle epoch) durante la discesa del gradiente. Utilizzando givens, possiamo conservare il dataset nella GPU, sempre se rientra nella sua capacità di memoria (per esempio se l'apprendimento avviene tramite mini-batch).

Il codice è il seguente:

```
# initialize
>>> data = np.array([[1, 2, 3]],
...                 dtype=theano.config.floatX)
>>> x = T.fmatrix('x')
>>> w = theano.shared(np.asarray([[0.0, 0.0, 0.0]],
...                              dtype=theano.config.floatX))
>>> z = x.dot(w.T)
>>> update = [[w, w + 1.0]]
# compile
>>> net_input = theano.function(inputs=[],
...                             updates=update,
...                             givens={x: data},
...                             outputs=z)
# execute
>>> for i in range(5):
...     print('z:', net_input())
z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]
```

Osservando questo esempio di codice, vediamo anche che l'attributo givens è un dizionario Python che mappa il nome di una variabile verso un effettivo oggetto Python. Qui impostiamo questo nome quando definiamo fmatrix.

Per riepilogare: un esempio di regressione lineare

Dopo aver familiarizzato con Theano, esaminiamo un esempio pratico: implementeremo in Theano la regressione *OLS* (*Ordinary Least Squares*). Per un breve ripasso dell'analisi a regressione, potete consultare il Capitolo 10, *Previsioni di variabili target continue: l'analisi a regressione*.

Iniziamo creando un piccolo dataset "giocattolo" monodimensionale, con cinque campioni per l'addestramento:

```
>>> X_train = np.asarray([[0.0], [1.0],
...                       [2.0], [3.0],
...                       [4.0], [5.0],
...                       [6.0], [7.0],
...                       [8.0], [9.0]],
...                      dtype=theano.config.floatX)
>>> y_train = np.asarray([1.0, 1.3,
...                       3.1, 2.0,
...                       5.0, 6.3,
...                       6.6, 7.4,
...                       8.0, 9.0],
...                      dtype=theano.config.floatX)
```

Notate che utilizziamo `theano.config.floatX` per costruire gli array NumPy, pertanto possiamo opzionalmente passare, indifferentemente, dall'utilizzo della CPU a quello della GPU, come desideriamo.

Poi implementiamo una funzione di addestramento per individuare i pesi del modello a regressione lineare, utilizzando la funzione di costo a somma degli errori al quadrato. Notate che w_0 è l'unità di bias (l'intercettazione dell'asse y avviene in $x = 0$). Il codice è il seguente:

```
import theano
from theano import tensor as T
import numpy as np
def train_linreg(X_train, y_train, eta, epochs):
    costs = []
    # Initialize arrays
    eta0 = T.fscalar('eta0')
    y = T.fvector(name='y')
    X = T.fmatrix(name='X')
    w = theano.shared(np.zeros(
                      shape=(X_train.shape[1] + 1),
                      dtype=theano.config.floatX),
                     name='w')

    # calculate cost
    net_input = T.dot(X, w[1:]) + w[0]
    errors = y - net_input
    cost = T.sum(T.pow(errors, 2))
    # perform gradient update
    gradient = T.grad(cost, wrt=w)
    update = [(w, w - eta0 * gradient)]
    # compile model
    train = theano.function(inputs=[eta0],
                        outputs=cost,
                        updates=update,
                        givens={X: X_train,
                                y: y_train,})

    for _ in range(epochs):
        costs.append(train(eta))

    return costs, w
```

Una funzionalità molto interessante di Theano è la funzione `grad` che abbiamo utilizzato nell'esempio di codice precedente. Tale funzione calcola automaticamente la derivata di un'espressione *rispetto ai suoi parametri* che abbiamo passato nell'argomento `wrt`.

Dopo aver implementato la funzione di addestramento, occupiamoci di addestrare il nostro modello a regressione lineare e diamo un'occhiata ai valori della funzione di costo *SSE* (*Sum of Squared Errors*) per controllare se converge:

```
>>> import matplotlib.pyplot as plt
>>> costs, w = train_linreg(X_train, y_train, eta=0.001, epochs=10)
>>> plt.plot(range(1, len(costs)+1), costs)
>>> plt.tight_layout()
>>> plt.xlabel('Epoch')
>>> plt.ylabel('Cost')
>>> plt.show()
```

Come possiamo vedere nella Figura 13.1, l'algoritmo di apprendimento converge già alla quinta epoch.

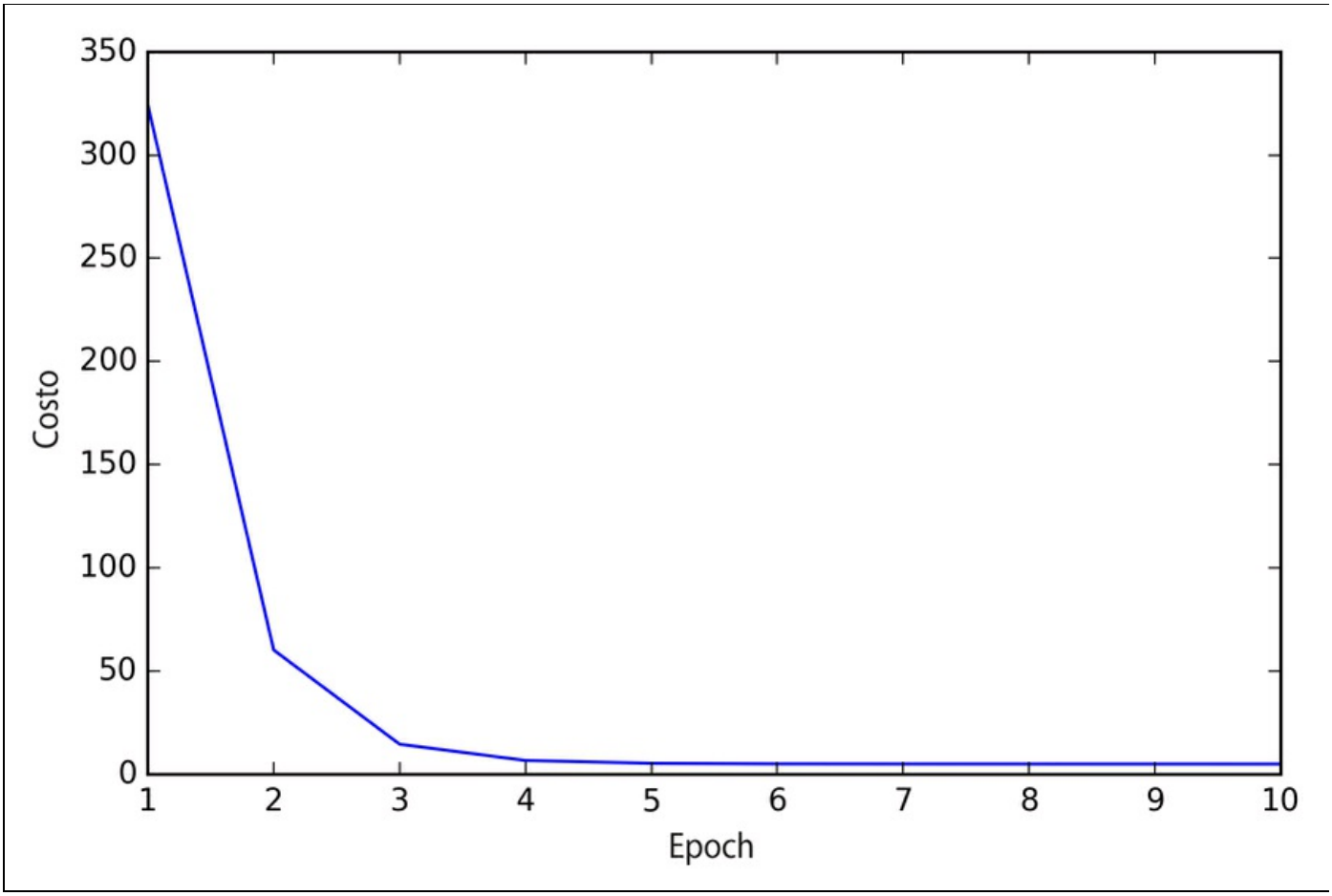


Figura 13.1

Finora niente male; osservando la funzione di costo, sembra che abbiamo costruito un modello a regressione funzionante per questo specifico dataset. Ora compiliamo una nuova funzione per eseguire delle previsioni sulla base delle caratteristiche di input:

```
def predict_linreg(X, w):
    Xt = T.matrix(name='X')
    net_input = T.dot(Xt, w[1:]) + w[0]
    predict = theano.function(inputs=[Xt],
                              givens={w: w},
                              outputs=net_input)
    return predict(X)
```

L'implementazione di una funzione `predict` è stata molto semplice, sulla base della procedura a tre passi di Theano: definizione, compilazione ed esecuzione. Ora tracciamo il risultato della regressione lineare sui dati di addestramento:

```
>>> plt.scatter(X_train,
...             y_train,
...             marker='s',
...             s=50)
>>> plt.plot(range(X_train.shape[0]),
```

Come possiamo vedere nella Figura 13.2, il nostro modello individua piuttosto bene tutti i punti dei dati.

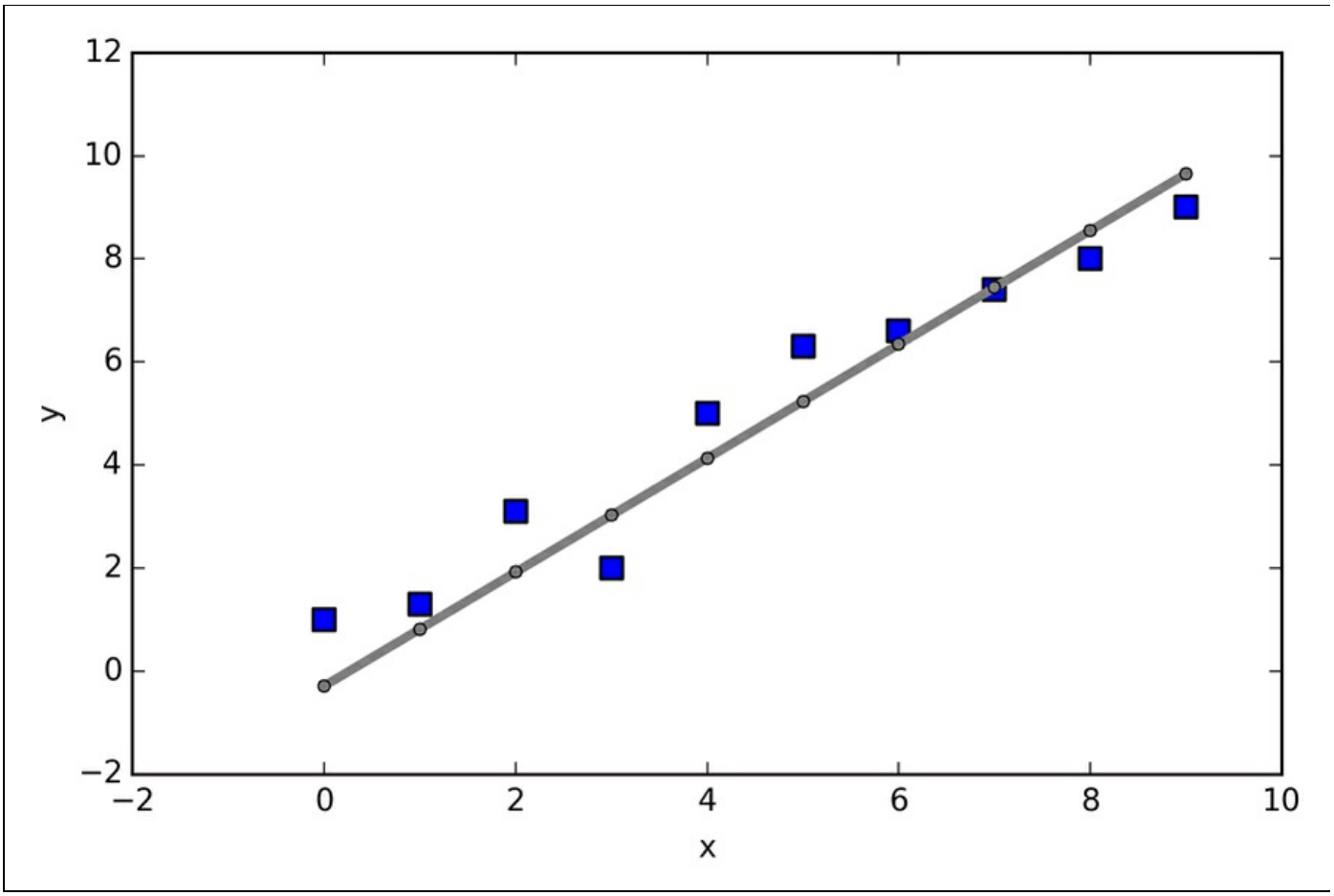


Figura 13.2

L'implementazione di un semplice modello a regressione è stato un buon esercizio per familiarizzare con l'API Theano. Tuttavia, il nostro scopo finale è quello di sfruttare i vantaggi di Theano, ovvero l'implementazione di potenti reti neurali artificiali. A questo punto siamo dotati di tutti gli strumenti di cui abbiamo bisogno per implementare il perceptron multilivello del Capitolo 12, *Reti neurali*

artificiali per il riconoscimento delle immagini, utilizzando però Theano. Tuttavia, la cosa sarebbe piuttosto noiosa, non è vero? Pertanto, diamo un'occhiata a una delle mie preferite librerie di deep learning basate su Theano per rendere ancora più comoda la sperimentazione con le reti neurali. Ma prima di introdurre la libreria Keras parliamo delle varie scelte in termini di funzioni di attivazione nelle reti neurali, argomento del prossimo paragrafo.

Scelta delle funzioni di attivazione per reti neurali ad avanzamento

Per semplicità, finora abbiamo parlato solo della funzione di attivazione sigmoid, nel contesto delle reti neurali multilivello ad avanzamento; l'abbiamo utilizzata nel livello nascosto e anche nel livello di output del perceptron multilivello che abbiamo implementato nel Capitolo 12, *Reti neurali artificiali per il riconoscimento delle immagini*. Sebbene abbiamo fatto riferimento a questa funzione di attivazione chiamando la funzione *sigmoid* (come è comunemente chiamata nella letteratura scientifica), una definizione più precisa sarebbe *funzione logistica* o *negative log-likelihood function*. Nei prossimi paragrafi parleremo di alcune altre funzioni sigmoidali alternative, che possono essere utili per implementare reti neurali multilivello.

Tecnicamente, potremmo utilizzare qualsiasi funzione (differenziabile) come funzione di attivazione di una rete neurale multilivello. Potremmo perfino utilizzare delle funzioni di attivazione lineari, come in Adaline (Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*). Tuttavia, nella pratica, non sarebbe molto utile impiegare funzioni di attivazione lineari per i livelli nascosti e di output, in quanto vogliamo introdurre la non linearità in una tipica rete neurale artificiale, in modo da essere in grado di affrontare anche problemi complessi. La somma di funzioni lineari, in fin dei conti, fornisce una funzione anch'essa lineare.

La funzione di attivazione logistica che abbiamo utilizzato nel capitolo precedente probabilmente emula più da vicino il comportamento di un neurone del cervello: possiamo considerarla come la probabilità che il neurone reagisca oppure no. Tuttavia, le funzioni di attivazione logistica possono essere problematiche quando abbiamo input fortemente negativi, in quanto l'output della funzione sigmoid, in questo caso, sarebbe molto vicino allo 0. Se la funzione sigmoid restituisce un output prossimo allo 0, la rete neurale comincerà a imparare molto lentamente e avrà maggiori probabilità di cadere vittima dei minimi locali durante l'addestramento. Questo è il motivo per cui molti preferiscono utilizzare una *tangente iperbolica* come funzione di attivazione dei livelli nascosti. Prima di parlare dell'aspetto di una tangente iperbolica, ricapitoliamo brevemente alcune delle basi della funzione logistica e osserviamo una generalizzazione particolarmente utile per compiti di classificazione multiclasse.

Ripasso sulla funzione logistica

Come abbiamo detto nell'introduzione a questo paragrafo, la funzione logistica, spesso chiamata *funzione sigmoid*, che in realtà è esattamente un caso speciale di una funzione sigmoid. Ricordiamo dal paragrafo dedicato alla regressione logistica del Capitolo 3, *I classificatori di machine learning di scikit-learn*, che in un compito di classificazione binaria possiamo utilizzare la funzione logistica per modellare la probabilità che il campione *x* appartenga alla classe positiva (classe 1):

$$\phi_{logistic}(z) = \frac{1}{1+e^{-z}}$$

Qui, la variabile scalare *z* è definita come l'input della rete:

$$z = w_0 x_0 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^T \boldsymbol{x}$$

Notate che w_0 è l'unità di bias (intercettazione dell'asse y, $x_0 = 1$). Per trattare un esempio più concreto, supponiamo l'utilizzo di un modello per un punto di dati bidimensionale x e un modello con i seguenti coefficienti di peso assegnati al vettore *w*:

```
>>> X = np.array([[1, 1.4, 1.5]])
>>> w = np.array([0.0, 0.2, 0.4])
>>> def net_input(X, w):
...     z = X.dot(w)
...     return z
>>> def logistic(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> def logistic_activation(X, w):
...     z = net_input(X, w)
...     return logistic(z)
>>> print('P(y=1|x) = %.3f'
...       % logistic_activation(X, w)[0])
P(y=1|x) = 0.707
```

Se calcolassimo l'input della rete e lo utilizzassimmo per attivare un neurone logistico con questi specifici valori della caratteristica e coefficienti di peso, otterremo un valore pari a 0.707, che possiamo interpretare come una probabilità del 70.7% che questo specifico campione *x* appartenga alla classe positiva. Nel Capitolo 12, *Reti neurali artificiali per il riconoscimento delle immagini*, abbiamo utilizzato la tecnica di codifica one-hot per calcolare i valori del livello di output costituito da più unità di attivazione logistica. Tuttavia, come dimostreremo con il seguente esempio di codice, un livello di output costituito da più unità di attivazione logistica non produce valori di probabilità significativi e interpretabili:

```
# W : array, shape = [n_output_units, n_hidden_units+1]
#         Weight matrix for hidden layer -> output layer.
```

```
# note that first column (A[:][0] = 1) are the bias units
>>> W = np.array([[1.1, 1.2, 1.3, 0.5],
...               [0.1, 0.2, 0.4, 0.1],
...               [0.2, 0.5, 2.1, 1.9]])
# A : array, shape = [n_hidden+1, n_samples]
#         Activation of hidden layer.
# note that first element (A[0][0] = 1) is the bias unit
>>> A = np.array([[1.0],
...               [0.1],
...               [0.3],
...               [0.7]])
# Z : array, shape = [n_output_units, n_samples]
#         Net input of the output layer.
>>> Z = W.dot(A)
>>> y_probas = logistic(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
 [[ 0.87653295]
 [ 0.57688526]
 [ 0.90114393]]
```

Come possiamo vedere dall'output, la probabilità che questo specifico campione appartenga alla prima classe è quasi dell'88%, ma la probabilità che questo campione appartenga alla seconda classe è quasi del 58% e la probabilità che quello stesso campione appartenga alla terza classe è del 90%. Questo non ha alcun senso, poiché sappiamo che una percentuale dovrebbe essere teoricamente espressa come una frazione del 100%. Tuttavia, questo non è davvero un grosso problema, se utilizziamo il nostro modello solo per prevedere le etichette delle classi e non le probabilità di appartenenza alla classe.

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('predicted class label: %d' % y_class[0])
predicted class label: 2
```

Ma in alcuni contesti, può essere utile restituire valori di probabilità di appartenenza più comprensibili per le previsioni multiclasse. Nel prossimo paragrafo esamineremo una generalizzazione della funzione logistica, la funzione *softmax*, che può essere utile per questo tipo di compiti.

Stima delle probabilità in un problema di classificazione multiclasse tramite la funzione softmax

La funzione *softmax* è una generalizzazione della funzione logistica che ci consente di calcolare le probabilità di appartenenza a una classe in un problema multiclasse (regressione logistica multinominale). In softmax, la probabilità di un determinato campione con input della rete z appartenga alla classe i-esima può essere calcolata con un termine di normalizzazione al denominatore che è la somma di tutte le funzioni lineari M:

$$P(y=i \mid z)=\phi_{softmax}(z)=\frac{e_i^z}{\sum_{m=1}^{M} e_m^z}$$

Per vedere in azione la funzione softmax, programmiamola in Python:

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))
>>> def softmax_activation(X, w):
...     z = net_input(X, w)
...     return softmax(z)
>>> y_probas = softmax(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
 [[ 0.40386493]
 [ 0.07756222]
 [ 0.51857284]]
>>> y_probas.sum()
1.0
```

Come possiamo vedere, le probabilità previste delle classi hanno ora una somma che vale 1, come ci aspetteremmo. è anche degno di nota il fatto che la probabilità per la seconda classe è prossima allo 0, poiché è troppa la distanza fra z_1 e $\max(z)$. Tuttavia, notate che l'etichetta della classe prevista è la stessa della funzione logistica. Intuitivamente, ha senso considerare la funzione softmax come una funzione logistica *normalizzata*, utile per ottenere previsioni sensate di appartenenza alle classi in una situazione multiclasse.

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('predicted class label:
...       %d' % y_class[0])
predicted class label: 2
```

Allargamento dello spettro di output utilizzando una tangente iperbolica

Un'altra funzione sigmoid che viene frequentemente utilizzata nei livelli nascosti delle reti neurali artificiali è la *tangente iperbolica* (*tanh*), che può essere interpretata come una versione in scala di una funzione logistica.

$$\phi_{\tanh}(z)=2\times\phi_{logistic}(2\times z)-1=\frac{e^z-e^{-z}}{e^z+e^{-z}}$$

$$\phi_{logistic}(z)=\frac{1}{1+e^{-z}}$$

Il vantaggio di una tangente iperbolica rispetto alla funzione logistica è il fatto che offre uno spettro di output più ampio e spazia nell'intervallo aperto (-1, 1), che

può migliorare la convergenza dell'algoritmo a retropropagazione (C. M. Bishop, *Neural networks for pattern recognition*, Oxford University Press, 1995, pp. 500-501). Al contrario, la funzione logistica restituisce un segnale di output nell'intervallo aperto (0, 1). Per un confronto intuitivo della funzione logistica e della tangente iperbolica, tracciamo due funzioni sigmoid in uno spazio monodimensionale:

```
>>> import matplotlib.pyplot as plt
>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)
>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)
>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('net input $z$')
>>> plt.ylabel('activation $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle='--')
>>> plt.axhline(0.5, color='black', linestyle='--')
>>> plt.axhline(0, color='black', linestyle='--')
>>> plt.axhline(-1, color='black', linestyle='--')
>>> plt.plot(z, tanh_act,
...          linewidth=2,
...          color='black',
...          label='tanh')
>>> plt.plot(z, log_act,
...          linewidth=2,
...          color='lightgreen',
...          label='logistic')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

Come possiamo vedere dalla Figura 13.3, la forma delle due curve sigmoidali ha un andamento simile; tuttavia, la funzione tanh offre uno spazio di output due volte superiore rispetto alla funzione logistica.

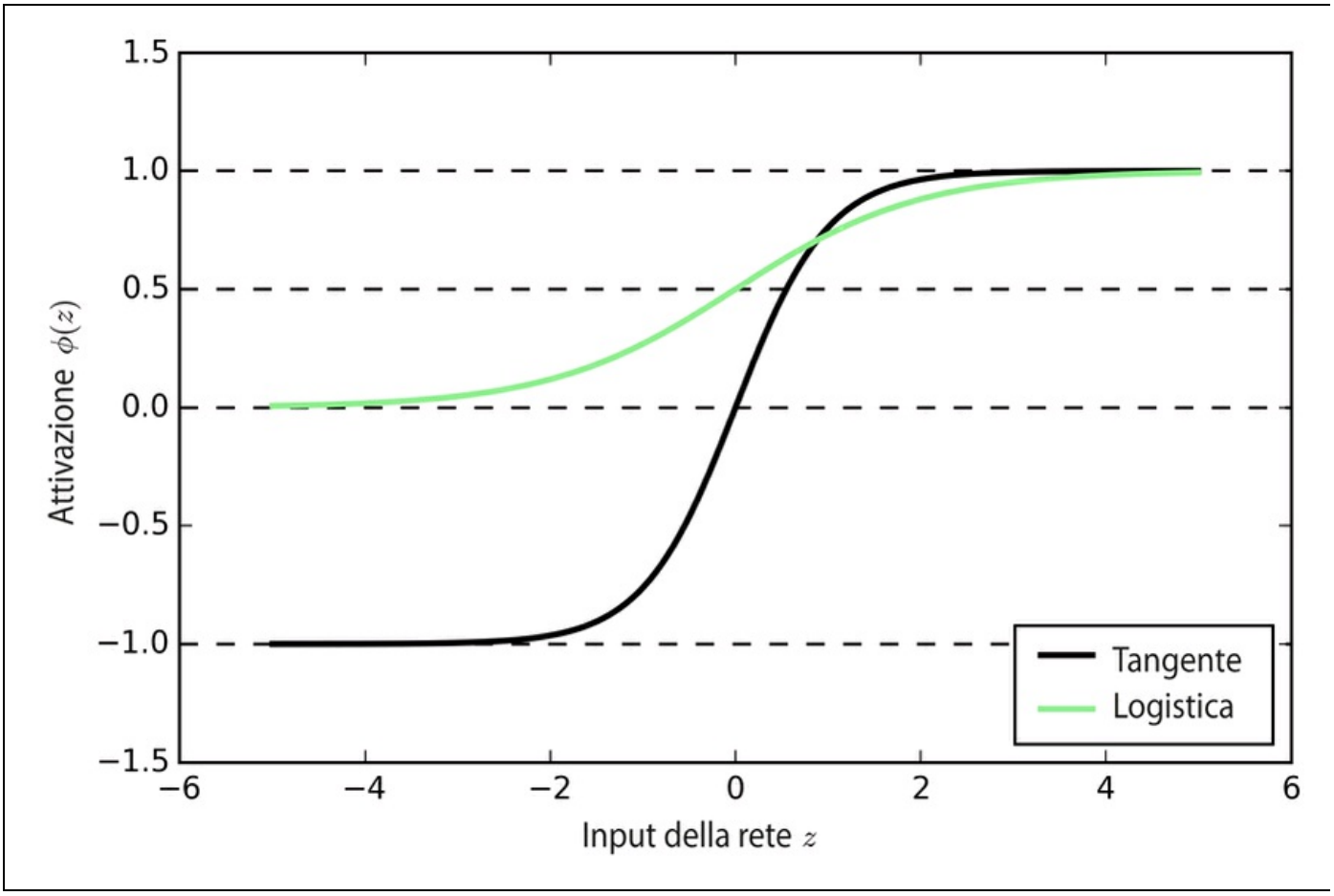


Figura 13.3

Notate che abbiamo implementato le funzioni `logistic` e `tanh` in modo prolisso per migliorare la loro rappresentatività. Nella pratica, possiamo utilizzare la funzione `tanh` di NumPy per ottenere gli stessi risultati:

```
>>> tanh_act = np.tanh(z)
```

Inoltre, la funzione `logistic` è disponibile nel modulo `special` di SciPy:

```
>>> from scipy.special import expit
>>> log_act = expit(z)
```

Ora che conosciamo qualcosa di più sulle varie funzioni di attivazione comunemente utilizzate nelle reti neurali artificiali, concludiamo il paragrafo con una panoramica delle diverse funzioni di attivazione che abbiamo incontrato in questo libro (Figura 13.4).

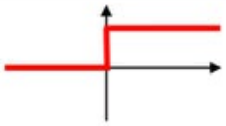
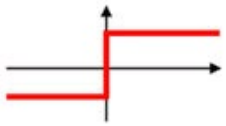
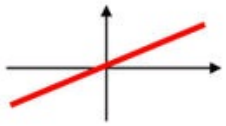
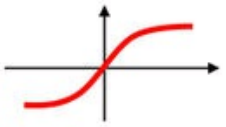
Funzione di attivazione	Equazione	Esempio	Grafico monodimensionale
Passo unitario (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Variante del perceptron	
Segno (Sigmun)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Variante del perceptron	
Lineare	$\phi(z) = z$	Adaline, regressione lineare	
Lineare a parti	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Macchina vettoriale di supporto	
Logistica (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Regressione logistica, rete neurale neurale multilivello	
Tangente iperbolica	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Rete neurale multilivello	

Figura 13.4

Addestramento efficiente delle reti neurali con Keras

In questo paragrafo esamineremo Keras, una delle librerie più recenti sviluppata per facilitare l'addestramento di reti neurali. Lo sviluppo di Keras è iniziato nei primi mesi del 2015. Attualmente Keras è diventata una delle librerie più note e ampiamente utilizzate basate su Theano che consente di utilizzare la GPU per accelerare l'addestramento delle reti neurali. Una delle sue principali caratteristiche è il fatto di essere un'API molto intuitiva, che ci consente di implementare le reti neurali con poche righe di codice. Dopo aver installato Theano, potete installare Keras da PyPI lanciando il seguente comando dal Terminale:

```
pip install Keras
```

Per ulteriori informazioni su Keras, visitate il suo sito web ufficiale all'indirizzo http://keras.io.

Per vedere come funziona l'addestramento di reti neurali tramite Keras, implementiamo un perceptron multilivello per classificare le cifre scritte a mano del dataset MNIST, che abbiamo introdotto nel capitolo precedente. Il dataset MNIST può essere scaricato da http://yann.lecun.com/exdb/mnist/ nelle quattro parti elencate di seguito.

- train-images-idx3-ubyte.gz: immagini del set di addestramento (9.912.422 byte)
- train-labels-idx1-ubyte.gz: etichette del set di addestramento (28.881 bytes)
- t10k-images-idx3-ubyte.gz: immagini del set di test (1.648.877 bytes)
- t10k-labels-idx1-ubyte.gz: etichette del set di test (4.542 bytes)

Dopo aver scaricato ed espanso gli archivi, collochiamo i file nella directory mnist della nostra directory di lavoro corrente, in modo da poter caricare il dataset di addestramento e quello di test utilizzando la seguente funzione:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte'
                               % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte'
                               % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
```

```
                dtype=np.uint8)
    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                imgpath.read(16))
        images = np.fromfile(imgpath,
                dtype=np.uint8).reshape(len(labels), 784)
    return images, labels
X_train, y_train = load_mnist('mnist', kind='train')
print('Rows: %d, columns: %d' % (X_train.shape[0], X_train.shape[1]))
Rows: 60000, columns: 784
X_test, y_test = load_mnist('mnist', kind='t10k')
print('Rows: %d, columns: %d' % (X_test.shape[0], X_test.shape[1]))
Rows: 10000, columns: 784
```

Nelle prossime pagine, esamineremo passo dopo passo alcuni esempi di codice per l'uso di Keras, che è possibile eseguire direttamente dall'interprete Python. Tuttavia, se siete interessati all'addestramento della rete neurale su GPU, potete inserirle in uno script Python oppure scaricare il codice degli esempi disponibile online. Per poter eseguire lo script Python sulla GPU, lanciate il seguente comando dalla directory in cui è situato il file mnist_keras_mlp.py:

```
THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python mnist_keras_mlp.py
```

Per continuare con la preparazione dei dati di addestramento, convertiamo l'array di immagini MNIST in formato a 32 bit:

```
>>> import theano
>>> theano.config.floatX = 'float32'
>>> X_train = X_train.astype(theano.config.floatX)
>>> X_test = X_test.astype(theano.config.floatX)
```

Poi dobbiamo convertire le etichette delle classi (interi compresi fra 0 e 9) nel formato one-hot. Fortunatamente, Keras ci offre un comodo strumento per farlo:

```
>>> from keras.utils import np_utils
>>> print('First 3 labels: ', y_train[:3])
First 3 labels:  [5 0 4]
>>> y_train_ohe = np_utils.to_categorical(y_train)
>>> print('\nFirst 3 labels (one-hot):\n', y_train_ohe[:3])
First 3 labels (one-hot):
 [[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]]
```

Ora possiamo occuparci della parte interessante e implementare una rete neurale. Utilizzeremo la stessa architettura già impiegata nel Capitolo 12, *Reti neurali artificiali per il riconoscimento delle immagini*. Tuttavia, sostituiremo le unità logistiche del livello nascosto con le funzioni di attivazione a tangente iperbolica, sostituiremo la funzione logistica del livello di output con una softmax e aggiungeremo un ulteriore livello nascosto. Keras rende queste operazioni molto semplici, come potete vedere dalla seguente implementazione di codice:

```
>>> from keras.models import Sequential
>>> from keras.layers.core import Dense
>>> from keras.optimizers import SGD
>>> np.random.seed(1)
>>> model = Sequential()
>>> model.add(Dense(input_dim=X_train.shape[1],
...                 output_dim=50,
...                 init='uniform',
...                 activation='tanh'))
>>> model.add(Dense(input_dim=50,
```

```
...             output_dim=50,
...             init='uniform',
...             activation='tanh'))
>>> model.add(Dense(input_dim=50,
...             output_dim=y_train_ohe.shape[1],
...             init='uniform',
...             activation='softmax'))
>>> sgd = SGD(lr=0.001, decay=1e-7, momentum=.9)
>>> model.compile(loss='categorical_crossentropy', optimizer=sgd)
```

Innanzitutto, inizializziamo un nuovo modello utilizzando la classe `Sequential` per implementare una rete neurale ad avanzamento ad avanzamento (*feed-forward*). Poi gli possiamo aggiungere tutti i livelli che desideriamo. Tuttavia, poiché il primo livello che aggiungiamo è il livello di input, dobbiamo assicurarci che l'attributo `input_dim` corrisponda al numero di caratteristiche (colonne) del set di addestramento (in questo caso sono ben 768). Inoltre, dobbiamo assicurarci che il numero di unità di output (`output_dim`) e di unità di input (`input_dim`) di due livelli consecutivi corrispondano fra loro. Nell'esempio precedente, abbiamo aggiunto due livelli nascosti con 50 unità nascoste, più un'unità di bias per ciascuno. Notate che le unità di bias sono inizializzate a 0 all'interno delle reti completamente connesse di Keras. Al contrario, nell'implementazione MLP presentata nel Capitolo 12, *Reti neurali artificiali per il riconoscimento delle immagini*, abbiamo inizializzato l'unità bias a 1, che è una convenzione più comune (ma non necessariamente la migliore).

Infine, il numero di unità del livello di output dovrebbe essere uguale al numero di etichette delle classi univoche (il numero di colonne nell'array di etichette delle classi in codifica one-hot). Prima di poter compilare il nostro modello, dobbiamo anche definire un ottimizzatore. Nell'esempio precedente, abbiamo scelto un'ottimizzazione con discesa del gradiente stocastica, che abbiamo già impiegato nei capitoli precedenti. Inoltre, possiamo impostare i valori per la costante di decadimento dei pesi e per il momentum learning, per regolare il tasso di apprendimento a ciascuna epoch, come discusso nel Capitolo 12, *Reti neurali artificiali per il riconoscimento delle immagini*. Infine, abbiamo impostato la funzione di costo (o delle perdite) su `categorical_crossentropy`. La *cross-entropy* (binaria) è solo un termine tecnico per chiamare la funzione di costo nella regressione logistica e la *cross-entropy categorica* è la sua generalizzazione per le previsioni multiclasse tramite softmax. Dopo aver compilato il modello, possiamo addestrarlo richiamando il metodo `fit`. Qui, utilizziamo un gradiente stocastico mini-batch con 300 campioni di addestramento per batch (dimensioni del lotto). Addestriamo il MLP su 50 epoch e chiediamo di seguire l'ottimizzazione della funzione `cost` durante l'addestramento impostando `verbose=1`. Il parametro `validation_split` è particolarmente

comodo, in quanto riserva il 10% dei dati di addestramento (in questo caso 6000 campioni) per la convalida dopo ciascuna epoch, in modo che possiamo controllare già durante l'addestramento se il modello soffre di problemi di overfitting.

```
>>> model.fit(X_train,
...           y_train_ohe,
...           nb_epoch=50,
...           batch_size=300,
...           verbose=1,
...           validation_split=0.1,
...           show_accuracy=True)
Train on 54000 samples, validate on 6000 samples
Epoch 0
54000/54000 [==============================] - 1s - loss: 2.2290 - acc: 0.3592 - val_loss: 2.1094 - val_acc: 0.5342
Epoch 1
54000/54000 [==============================] - 1s - loss: 1.8850 - acc: 0.5279 - val_loss: 1.6098 - val_acc: 0.5617
Epoch 2
54000/54000 [==============================] - 1s - loss: 1.3903 - acc: 0.5884 - val_loss: 1.1666 - val_acc: 0.6707
Epoch 3
54000/54000 [==============================] - 1s - loss: 1.0592 - acc: 0.6936 - val_loss: 0.8961 - val_acc: 0.7615
[…]
Epoch 49
54000/54000 [==============================] - 1s - loss: 0.1907 - acc: 0.9432 - val_loss: 0.1749 - val_acc: 0.9482
```

La stampa del valore della funzione di costo è particolarmente utile durante l'addestramento, in quanto consente di individuare rapidamente se sta decrescendo durante l'addestramento e, in caso contrario, di fermare l'algoritmo anticipatamente, così da ottimizzarne i parametri.

Per prevedere le etichette delle classi, possiamo utilizzare il metodo `predict_classes`, che restituisce le etichette delle classi direttamente sotto forma di interi:

```
>>> y_train_pred = model.predict_classes(X_train, verbose=0)
>>> print('First 3 predictions: ', y_train_pred[:3])
>>> First 3 predictions:  [5 0 4]
```

Infine, stampiamo l'accuratezza del modello sui set di addestramento di test:

```
>>> train_acc = np.sum(
...       y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Training accuracy: %.2f%%' % (train_acc * 100))
Training accuracy: 94.51%
>>> y_test_pred = model.predict_classes(X_test, verbose=0)
>>> test_acc = np.sum(y_test == y_test_pred,
...                   axis=0) / X_test.shape[0]
print('Test accuracy: %.2f%%' % (test_acc * 100))
Test accuracy: 94.39%
```

Notate che questa è una rete neurale davvero semplice, senza parametri di configurazione ottimizzati. Se siete interessati a utilizzare approfonditamente Keras, intervenite anche sul tasso di apprendimento, il momentum, il decadimento dei pesi e il numero di unità nascoste.

NOTA

Sebbene Keras sia un'ottima libreria per l'implementazione e la sperimentazione con le reti neurali, esistono anche molte altre librerie wrapper per Theano che vale la pena di menzionare. Un esempio particolarmente interessante è Pylearn2 (http://deeplearning.net/software/pylearn2/), che è stato sviluppato ai laboratori LISA di Montreal. Inoltre, Lasagne (https://github.com/Lasagne/Lasagne) può essere particolarmente interessante se preferite una libreria minimalista ma espandibile, che offre un maggiore controllo sul codice Theano sottostante.

Riepilogo

Spero che abbiate apprezzato quest'ultimo capitolo con un eccitante tour nel campo del machine learning. Nel corso dell'intero libro, abbiamo trattato tutti gli argomenti essenziali di questa materia e ora dovreste essere dotati di tutte le conoscenze tecniche necessarie per risolvere i problemi presentati dal mondo reale.

Abbiamo iniziato il nostro viaggio con una breve panoramica dei vari tipi di compiti di apprendimento: apprendimento con supervisione, apprendimento con rafforzamento e apprendimento senza supervisione. Abbiamo parlato dei vari algoritmi di apprendimento che possono essere utilizzati per la classificazione, a partire dalle semplici reti neurali monolivello nel Capitolo 2, *Addestrare gli algoritmi a compiti di classificazione*. Poi abbiamo parlato di algoritmi di classificazione più avanzati, nel Capitolo 3, *I classificatori di machine learningdi scikit-learn*, e abbiamo scoperto i più importanti aspetti di una catena di machine learning nel Capitolo 4, *Costruire buoni set di addestramento: la pre-elaborazione*, e nel Capitolo 5, *Compressione dei dati tramite la riduzione della dimensionalità*. Ricordate che anche l'algoritmo più avanzato è limitato dalle informazioni presenti nei dati di addestramento che utilizza per imparare. Nel Capitolo 6, *Valutazione dei modelli e ottimizzazione degli iperparametri*, abbiamo conosciuto le tecniche migliori per costruire e valutare le modalità predittive, che sono un altro aspetto importante delle applicazioni di machine learning. Se un unico algoritmo di apprendimento non fornisce le prestazioni desiderate, può essere utile creare un insieme di esperti per eseguire una previsione. Ne abbiamo parlato nel Capitolo 7, *Combinare più modelli: l'apprendimento d'insieme*. Nel Capitolo 8, *Tecniche di machine learning per l'analisi del sentiment*, abbiamo applicato tecniche di machine learning all'analisi della forma di dati probabilmente più interessante al giorno d'oggi, che predomina nelle piattaforme dei social media in Internet: i documenti testuali. Tuttavia, le tecniche di machine learning non si limitano all'analisi dei dati offline e nel Capitolo 9, *Embedding di un modello in un'applicazione web*, abbiamo visto come incorporare un modello di machine learning in un'applicazione web, per condividerlo con il resto del mondo. Per lo più, ci siamo concentrati su algoritmi di classificazione, probabilmente l'applicazione più diffusa di machine learning. Tuttavia, le cose non finiscono qui. Nel Capitolo 10, *Previsioni di variabili target continue: l'analisi a regressione*, abbiamo esplorato vari algoritmi per l'analisi a regressione, per prevedere valori di

output su scala continua. Un'altra branca interessante del machine learning è l'analisi a cluster, che ci aiuta a trovare le strutture nascoste nei dati, anche quando tali dati non contengono le risposte corrette, con cui eseguire confronti e da cui imparare. Ne abbiamo parlato nel Capitolo 11, *Lavorare con dati senza etichette: l'analisi a cluster*.

Nei due capitoli finali del libro, abbiamo esaminato gli algoritmi più avanzati e interessanti dell'intero campo del machine learning: le reti neurali artificiali. Sebbene il deep learning non rientri del tutto negli scopi di questo libro, spero di aver risvegliato almeno il vostro interesse nel seguire i più recenti avanzamenti in questo campo. Se state considerando una carriera come ricercatori nell'ambito del machine learning o anche se volete tenervi aggiornati con gli attuali avanzamenti in questo campo, vi consiglio di seguire i lavori dei principali esperti in questo campo, come Geoff Hinton (http://www.cs.toronto.edu/~hinton/), Andrew Ng (http://www.andrewng.org), Yann LeCun (http://yann.lecun.com), Juergen Schmidhuber (http://people.idsia.ch/~juergen/) e Yoshua Bengio (http://www.iro.umontreal.ca/~bengioy), solo per nominarne alcuni. Inoltre, non esitate a abbonarvi alle mailing-list di scikit-learn, Theano e Keras, per partecipare a interessanti discussioni su queste librerie e sul machine learning in generale. Mi auguro proprio di ritrovarvi là!

Spero che questo viaggio tra i vari aspetti del machine learning vi abbia appassionato e che abbiate appreso molte tecniche aggiornate e utili per migliorare la vostra carriera e per la soluzione dei problemi.