Capitolo 1

Panoramica sugli algoritmi

> Un algoritmo deve essere visto all'opera per poter essere creduto.
>
> *Donald Knuth*

Questo libro tratta tutte le informazioni necessarie per comprendere, classificare, selezionare e implementare alcuni fra gli algoritmi più importanti. Oltre a spiegare la logica su cui essi si basano, il libro tratta anche di strutture di dati, ambienti di sviluppo e ambienti di produzione, spiegando quali sono più adatti alle diverse classi di algoritmi. Questa è la seconda edizione del libro. In questa edizione ci concentreremo specialmente sui moderni algoritmi di *machine learning*, che stanno diventando sempre più importanti, al giorno d'oggi. Insieme alla logica, esamineremo anche alcuni esempi pratici dell'uso di algoritmi per risolvere problemi di carattere quotidiano.

Questo capitolo offre una panoramica sui fondamenti degli algoritmi. Si apre con un paragrafo dedicato ai concetti di base necessari per comprendere il funzionamento dei diversi algoritmi. Per offrire una prospettiva storica, questo primo paragrafo riassume il modo in cui si è iniziato a utilizzare gli algoritmi per formulare matematicamente una certa classe di problemi e menziona anche i limiti dei diversi algoritmi. Il paragrafo successivo spiega i vari modi per specificare la logica di un algoritmo. Poiché in questo libro, per scrivere gli algoritmi viene utilizzato il linguaggio di programmazione Python, vedremo anche come impostare l'ambiente di programmazione adatto a svolgere gli esempi. Quindi, scopriremo

In questo capitolo

i vari modi in cui quantificare le prestazioni di un algoritmo, per poter confrontare fra loro più algoritmi. Infine, il capitolo discute vari metodi di convalida di una determinata implementazione di un algoritmo.

Che cos'è un algoritmo?

In breve, un *algoritmo* è un insieme di regole per eseguire i calcoli necessari per risolvere un problema. Un algoritmo è progettato per fornire risultati per qualsiasi input valido, secondo istruzioni definite con precisione. Cercando la definizione di algoritmo in un dizionario come l'American Heritage, si trova che:

> Un algoritmo è un insieme finito di istruzioni non ambigue che, dato un insieme di condizioni iniziali, può essere eseguito secondo una sequenza prescritta per raggiungere un determinato obiettivo e che ha un insieme riconoscibile di condizioni finali.

La progettazione di un algoritmo tenta di creare una ricetta matematica del modo più efficiente per risolvere un determinato problema concreto. Questa ricetta può essere utilizzata come base per sviluppare una soluzione matematica riutilizzabile e generica, che possa essere applicata a un insieme più ampio di problemi simili.

Le fasi di un algoritmo

La Figura 1.1 mostra le diverse fasi di sviluppo, implementazione e utilizzo di un algoritmo.

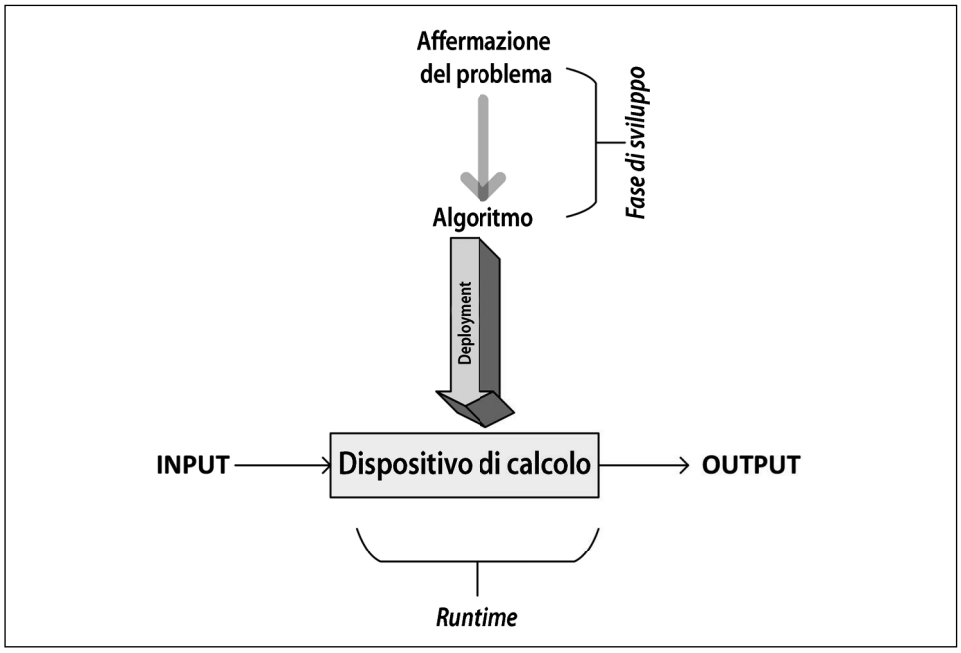

Figura 1.1 Le diverse fasi di sviluppo, deployment e utilizzo di un algoritmo.

Come possiamo vedere, il processo inizia con la comprensione dei requisiti, dall'affermazione del problema, che specificano quale risultato occorre ottenere. Una volta che il problema è esposto chiaramente, si passa alla fase di sviluppo.
La fase di sviluppo di un algoritmo si compone a sua volta di due fasi.

- La *fase di progettazione* duranre la quale vengono immaginati e documentati l'architettura, la logica e i dettagli di implementazione dell'algoritmo. In questa fase teniamo presente sia l'accuratezza sia le prestazioni. Durante la ricerca della soluzione migliore per un dato problema, in molti casi finiremo per avere più algoritmi "candidati". La fase di progettazione di un algoritmo è un processo iterativo, che prevede il confronto di diversi algoritmi candidati. Alcuni algoritmi possono fornire soluzioni semplici e veloci, ma a scapito della precisione. Altri algoritmi possono essere molto accurati, ma la loro esecuzione può richiedere molto tempo, a causa della loro complessità. Alcuni di questi algoritmi complessi possono essere più efficienti di altri. Prima di effettuare una scelta, occorre studiare attentamente tutti i compromessi intrinseci degli algoritmi candidati. Soprattutto se il problema è complesso, è importante progettare un algoritmo efficiente. Un algoritmo progettato correttamente si tradurrà in una soluzione efficiente che sarà in grado di fornire, allo stesso tempo, prestazioni soddisfacenti e un'accuratezza ragionevole.
- La *fase di programmazione* duranre la quale l'algoritmo progettato viene convertito in un programma. È importante che tale programma implementi tutte le logiche e le architetture suggerite in fase di progettazione.

I requisiti del problema in questione possono essere suddivisi in requisiti funzionali e non funzionali. I requisiti che specificano direttamente le caratteristiche delle soluzioni sono chiamati requisiti funzionali. I requisiti funzionali descrivono in dettaglio il comportamento previsto della soluzione. Al contrario, i requisiti non funzionali riguardano le prestazioni, la scalabilità, l'usabilità e l'accuratezza dell'algoritmo. I requisiti non funzionali stabiliscono anche le aspettative sulla sicurezza dei dati. Per esempio, immaginiamo di dover progettare un algoritmo per una società di carte di credito, che sia in grado di identificare e segnalare le transazioni fraudolente. I requisiti funzionali, in questo caso, specificheranno il comportamento previsto di una soluzione valida, fornendo i dettagli dell'output previsto dato un determinato insieme di dati di input. In questo caso, i dati di input potrebbero essere i dettagli della transazione e l'output potrebbe essere un flag binario che etichetta una transazione come fraudolenta oppure non fraudolenta. In questo esempio, i requisiti non funzionali possono specificare il tempo di risposta di ciascuna previsione. I requisiti non funzionali fisseranno anche le soglie consentite per la precisione. Poiché in questo esempio abbiamo a che fare con dati finanziari, si prevede che anche i requisiti di sicurezza relativi all'autenticazione dell'utente, all'autorizzazione e alla riservatezza dei dati facciano parte dei requisiti non funzionali.
Notate che i requisiti funzionali e non funzionali mirano a definire con precisione che cosa deve essere fatto. Progettare la soluzione significa capire come sarà realizzata. Implementare il progetto significa sviluppare la soluzione effettiva nel linguaggio di programmazione scelto. Elaborare un progetto che soddisfi pienamente i requisiti sia funzionali sia non funzionali può richiedere molto tempo e impegno. La scelta del giusto linguaggio di programmazione e dell'ambiente di sviluppo/produzione può dipendere dai requisiti del problema. Per esempio, poiché C/C++ è un linguaggio di livello inferiore rispetto a Python, potrebbe essere una scelta migliore per algoritmi che necessitano di codice compilato e di ottimizzazioni di livello più profondo.

Una volta completata la fase di progettazione e la programmazione, l'algoritmo è pronto per il deployment. Il deployment di un algoritmo implica la progettazione dell'ambiente di produzione effettivo in cui verrà eseguito il codice. L'ambiente di produzione deve essere progettato in base ai dati e alle esigenze di elaborazione dell'algoritmo. Per esempio, per gli algoritmi parallelizzabili, sarà necessario un cluster con un numero appropriato di nodi di calcolo per l'esecuzione efficiente dell'algoritmo. Per gli algoritmi ad alta intensità di dati, potrebbe essere necessario progettare una pipeline di ingresso dei dati e una strategia di gestione della cache e di archiviazione dei dati. La progettazione di un ambiente di produzione viene trattata in maggior dettaglio nei Capitoli 15 e 16.
Una volta progettato e implementato l'ambiente di produzione, l'algoritmo passa al deployment, che prende i dati di input, li elabora e genera l'output secondo i requisiti.

L'ambiente di sviluppo

Una volta progettati, gli algoritmi devono essere implementati in un linguaggio di programmazione in base al progetto. Per questo libro ho scelto di impiegare il linguaggio di programmazione Python. L'ho scelto perché Python è un linguaggio di programmazione flessibile e open source. Python è anche uno dei linguaggi preferiti per infrastrutture di cloud computing come *Amazon Web Services* (AWS), *Microsoft Azure* e *Google Cloud Platform* (GCP).
La home page ufficiale di Python è disponibile su `https://www.python.org`, che offre anche le istruzioni per l'installazione e un'utile guida rivolta ai principianti.
Una conoscenza di base di Python vi aiuterà a comprendere meglio i concetti presentati nel libro. In questo libro, mi aspetto che utilizziate la versione più recente di Python 3. Al momento della stesura di queste pagine, la versione più recente è la 3.10, che è quella che useremo per gli esercizi.
Utilizzeremo Python un po' in tutto il libro. Utilizzeremo anche Jupyter Notebook per eseguire il codice. I capitoli di questo libro presuppongono che Python sia installato e che Jupyter Notebook sia stato configurato correttamente e sia in esecuzione.

Pacchetti Python

Python è un linguaggio *general-purpose*, ovvero "di uso generale". Segue la filosofia del "batterie incluse", il che significa che rende disponibile una libreria standard, senza che l'utente debba scaricare altri pacchetti. Tuttavia, i moduli della libreria standard forniscono solo le funzionalità di base. In base al caso d'uso specifico su cui state lavorando, potrebbe essere necessario installare altri pacchetti. Il repository ufficiale di terze parti per i pacchetti Python si chiama PyPI, che sta per *Python Package Index*. Ospita pacchetti Python sia come codice sorgente sia come codice precompilato. Attualmente, PyPI ospita più di 113.000 pacchetti Python. Il modo più semplice per installare pacchetti aggiuntivi è tramite il sistema di gestione dei pacchetti `pip`, un acronimo ricorsivo nerd, che abbonda nella cultura Python, e che sta per "Pip Installs Python". La buona notizia è che a partire dalla versione 3.4 di Python, `pip` è installato di default. Per verificare la versione di pip, potete digitare il seguente comando sulla riga di comando:

```
pip --version
```

Ecco, invece, il comando `pip` da usare per installare nuovi pacchetti:

```
pip install NomePacchetto
```

I pacchetti precedentemente installati devono essere aggiornati periodicamente per ottenere le ultime funzionalità. Per fare ciò si utilizza il flag `upgrade`:

```
pip install NomePacchetto --upgrade
```

E anche per installare una determinata versione di un pacchetto Python:

```
pip install NomePacchetto==2.1
```

NOTA

L'aggiunta delle librerie e delle versioni corrette è diventata parte della configurazione dell'ambiente di programmazione Python. Una funzionalità che aiuta a gestire queste librerie consiste nella possibilità di creare un file dei requisiti che elenca tutti i pacchetti necessari. Il file dei requisiti è un semplice file di testo che contiene il nome delle librerie e le relative versioni. Per esempio, il file dei requisiti può avere il seguente aspetto:

```
scikit-learn==0.24.1
tensorflow==2.5.0
tensorboard==2.5.0
```

Per convenzione, il file `requirements.txt` viene collocato nella directory superiore del progetto.
Una volta creato, il file dei `requirements.txt` può essere utilizzato per configurare l'ambiente di sviluppo installando tutte le librerie Python e le relative versioni utilizzando il seguente comando:

```
pip install -r requirements.txt
```

Consideriamo ora i principali pacchetti che utilizzeremo in questo libro.

L'ecosistema SciPy

Scientific Python (SciPy) è un gruppo di pacchetti Python creati appositamente per la comunità scientifica. Contiene molte funzioni, fra cui un'ampia gamma di generatori di numeri casuali, numerose routine di algebra lineare e svariati ottimizzatori.
SciPy è un pacchetto completo e, nel tempo, molti hanno sviluppato varie estensioni per personalizzare ed estendere il pacchetto in base alle proprie esigenze. SciPy è un pacchetto ad alte prestazioni, poiché "avvolge" codice ottimizzato scritto in C/C++ o Fortran.
Ecco i principali pacchetti che compongono questo ecosistema.

- *NumPy*: per gli algoritmi, la capacità di creare strutture di dati multidimensionali, come array e matrici, è davvero importante. NumPy offre una serie di tipi di dati per array e matrici utili per i calcoli statistici e l'analisi dei dati. Per informazioni su NumPy: `http://www.numpy.org`.

- *scikit-learn*: questa estensione, dedicata al machine learning, è una delle estensioni più utilizzate di SciPy. scikit-learn offre un'ampia gamma di utili algoritmi di machine learning, fra cui quelli di classificazione, regressione, clustering e convalida dei modelli. Per informazioni su scikit-learn: `http://scikit-learn.org`.
- *pandas*: è una libreria software che contiene un'articolata struttura di dati tabulare ampiamente utilizzata per operazioni di input, output ed elaborazione di dati tabulari in vari algoritmi. La libreria pandas contiene molte funzioni utili e offre prestazioni altamente ottimizzate. Per informazioni su pandas: `http://pandas.pydata.org`.
- *Matplotlib*: fornisce numerosi strumenti per creare visualizzazioni di grande impatto. I dati possono essere presentati come grafici a linee, grafici a dispersione, grafici a barre, istogrammi, grafici a torta e così via. Per informazioni su Matplotlib: `https://matplotlib.org`.

Utilizzo di Jupyter Notebook

Utilizzeremo come IDE (ambiente di sviluppo) Jupyter Notebook e Google Colaboratory. Maggiori dettagli sulla configurazione e sull'uso di Jupyter Notebook e Colab sono disponibili nelle Appendici A e B.

Tecniche di progettazione degli algoritmi

Un algoritmo è una soluzione matematica a un problema concreto. Quando progettiamo un algoritmo, teniamo a mente i seguenti tre problemi di progettazione.

1. Questo algoritmo sta producendo il risultato che ci aspettavamo?
2. Questo algoritmo rappresenta il modo ottimale per ottenere questi risultati?
3. Come funzionerà l'algoritmo su dataset di maggiori dimensioni?

È importante studiare a fondo la complessità del problema, prima di progettarne una soluzione. Per esempio, per progettare una soluzione adeguata è utile caratterizzare il problema in termini di sue esigenze e complessità.
In generale, gli algoritmi possono essere suddivisi nelle seguenti tipologie, in base alle caratteristiche del problema.

- *Algoritmi a elevata intensità di dati* (*data-intensive*): sono progettati per gestire grandi quantità di dati. Ci si aspetta che abbiano requisiti di elaborazione relativamente ridotti. Un algoritmo di compressione applicato a un file enorme è un buon esempio di algoritmo a elevata intensità di dati. Per tali algoritmi, la dimensione dei dati gestibili dovrebbe essere molto maggiore della memoria del sistema di elaborazione (il singolo nodo o l'intero cluster) e, in base ai requisiti, potrebbe essere necessario adottare uno schema di elaborazione iterativo per poter elaborare in modo efficiente i dati.
- *Algoritmi a elevata intensità di calcoli* (*compute-intensive*): hanno requisiti di elaborazione considerevoli, ma non elaborano grandi quantità di dati. Un semplice esempio è l'algoritmo per trovare un numero primo molto grande. Per massimizzare le prestazioni dell'algoritmo è fondamentale trovare una strategia per suddividere l'algoritmo in più fasi, in modo che almeno alcune di esse possano essere parallelizzate.

- *Algoritmi a elevata intensità di dati e di calcoli*: esistono alcuni algoritmi che si trovano a gestire una grande quantità di dati ma hanno anche notevoli requisiti di in termini di elaborazione. Gli algoritmi utilizzati per eseguire l'analisi del *sentiment* per i feed video in diretta sono un buon esempio di situazioni in cui sia i dati sia i requisiti di elaborazione sono enormi, per poter portare a termine l'attività. Questi sono gli algoritmi più dispendiosi in termini di risorse e richiedono un'attenta progettazione e un'allocazione intelligente delle risorse disponibili.

Per caratterizzare il problema in termini di complessità e requisiti, è utile studiare i dati e calcolare le dimensioni in modo più approfondito, cosa che faremo nel prossimo paragrafo.

La dimensione dei dati

Per classificare la dimensione dei dati del problema, osserviamo il loro volume, la loro velocità e la loro varietà (le tre "V"), che sono definite come segue.

- *Volume*: è la dimensione fisica prevista dei dati che l'algoritmo dovrà elaborare.
- *Velocità*: è la rapidità con la quale vengono generati nuovi dati quando viene utilizzato l'algoritmo. Può essere pari a zero.
- *Varietà*: descrive quanti diversi tipi di dati l'algoritmo si troverà a elaborare.

La Figura 1.2 mostra più in dettaglio le tre "V" dei dati. Al centro di questo grafico si trovano i dati più semplici possibili: piccolo volume, scarsa varietà e bassa velocità. A mano a mano che ci allontaniamo dal centro, la complessità dei dati aumenta, e può farlo in una o più delle tre dimensioni. Per esempio, nella dimensione della velocità, abbiamo il semplice processo batch, seguito dal processo periodico e quindi dal processo near real-time; infine abbiamo il processo real-time, che è il più complesso da gestire in termini di velocità dei dati. Per esempio, la raccolta dei feed video live prodotti da un gruppo di telecamere di monitoraggio avrà un grande volume, un'elevata velocità e una notevole varietà e per poter archiviare ed elaborare i dati in modo efficace sarà necessario realizzare un progetto appropriato.
Consideriamo tre esempi di casi d'uso con tre diversi tipi di dati.

- Innanzitutto, consideriamo un semplice caso d'uso in cui i dati di input sono in un file `.csv`. In questo caso, il volume, la velocità e la varietà dei dati saranno bassi.
- In secondo luogo, consideriamo il caso d'uso in cui i dati di input sono forniti dal flusso live di una videocamera di sicurezza. Ora il volume, la velocità e la varietà dei dati saranno piuttosto elevati e dovrebbero essere tenuti in considerazione durante la progettazione di un algoritmo per gestirli.
- In terzo luogo, consideriamo il caso d'uso di una tipica rete di sensori. Supponiamo che la fonte sia una rete di sensori di temperatura installati in un grande edificio. Sebbene la velocità dei dati generati sia tipicamente molto elevata (poiché i nuovi dati vengono generati molto rapidamente), si prevede che il volume sarà piuttosto basso (poiché ciascun elemento di dati è tipicamente lungo solo 16 bit e consiste di 8 bit di misurazione più 8 bit di metadati, come il timestamp e le coordinate geografiche).

I requisiti di elaborazione, le esigenze di archiviazione e la selezione dello stack software adeguato saranno diversi per tutti e tre gli esempi precedenti e, in generale, dipenderanno dal volume, dalla velocità e dalla varietà delle origini dei dati. Come primo passaggio nella progettazione di un algoritmo, è importante, innanzitutto, caratterizzare i dati.

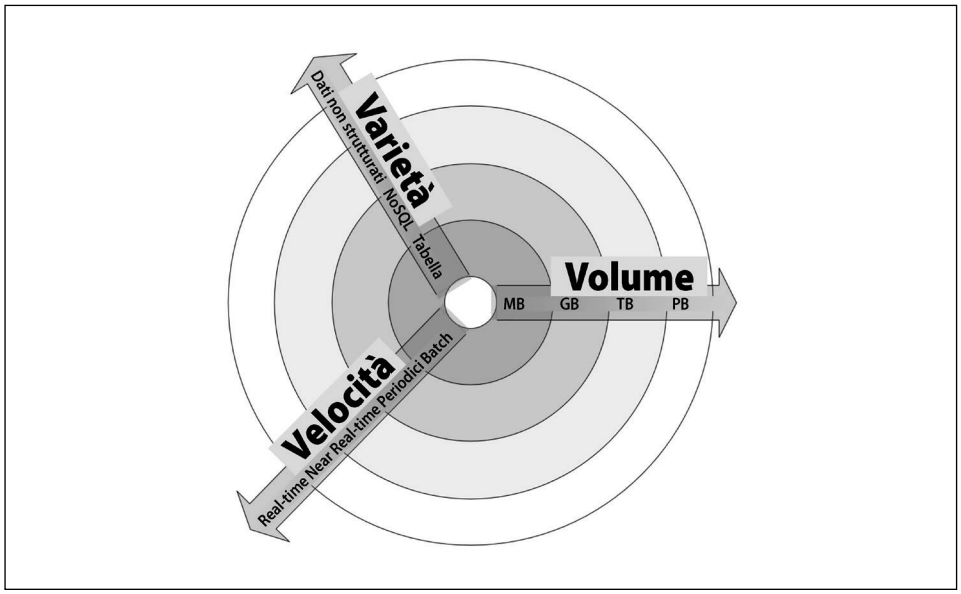

Figura 1.2 Le 3V dei dati: Volume, Velocità e Varietà.

La dimensione del calcolo

Per caratterizzare la dimensione del calcolo, analizziamo le esigenze di elaborazione del problema in questione. Le esigenze di elaborazione di un algoritmo determinano quale tipo di progettazione è più efficiente. Per esempio, gli algoritmi complessi, in generale, richiedono molta potenza di calcolo. Per tali algoritmi, potrebbe essere importante impiegare un'architettura parallela multinodo. I moderni algoritmi "deep", di solito impiegano una notevole elaborazione numerica e potrebbero richiedere la potenza di GPU o TUP, come discusso nel Capitolo 16.

Analisi delle prestazioni

L'analisi delle prestazioni di un algoritmo è un aspetto importante della sua progettazione. Uno dei modi per stimare le prestazioni di un algoritmo è analizzarne la complessità. La *teoria della complessità* studia il "peso" degli algoritmi. Per essere davvero utile, un buon algoritmo deve avere tre caratteristiche chiave.

- *Deve essere corretto*: un buon algoritmo deve produrre il risultato corretto. Per confermare il fatto che un algoritmo funziona correttamente, occorre sottoporlo a rigorosi test, specialmente sui casi d'uso.
- *Deve essere comprensibile*: il miglior algoritmo del mondo non servirà a niente se è troppo complicato per poter essere implementato su un computer.
- *Deve essere efficiente*: un algoritmo che produca un risultato corretto e che sia implementabile non vi aiuterà molto se impiega mille anni o se richiede 1 miliardo di terabyte di memoria per fornire il suo risultato.

Esistono due possibili tipi di analisi per quantificare la complessità di un algoritmo.

- *Analisi della complessità spaziale*: stima i requisiti di memoria a runtime necessari per eseguire l'algoritmo.
- *Analisi della complessità temporale*: stima il tempo necessario per l'esecuzione dell'algoritmo.

Analisi della complessità spaziale

Stima la quantità di memoria richiesta dall'algoritmo per elaborare i dati di input. Durante l'elaborazione dei dati di input, l'algoritmo deve memorizzare alcune strutture di dati temporanee. Il modo in cui l'algoritmo è progettato influenza il numero, il tipo e la dimensione di queste strutture di dati. In questa era di calcolo distribuito e con quantità sempre più grandi di dati da elaborare, l'analisi della complessità spaziale sta diventando sempre più importante. La dimensione, il tipo e il numero di queste strutture di dati determinano i requisiti di memoria per l'hardware sottostante. Le moderne strutture di dati in memoria utilizzate nel calcolo distribuito devono disporre di meccanismi efficienti di allocazione delle risorse, che siano consapevoli dei requisiti di memoria nelle diverse fasi di esecuzione dell'algoritmo.
Gli algoritmi complessi tendono a essere di natura iterativa. Invece di portare le informazioni in memoria tutte in una volta, tali algoritmi popolano in modo iterativo le strutture di dati. Per calcolare la complessità spaziale, è importante innanzitutto classificare il tipo di algoritmo iterativo che intendiamo utilizzare. Un algoritmo iterativo può utilizzare uno dei tre tipi seguenti di iterazioni.

- *Iterazioni convergenti*: man mano che l'algoritmo svolge le iterazioni, la quantità di dati elaborati in ogni singola iterazione diminuisce. In altre parole, la complessità spaziale diminuisce man mano che l'algoritmo procede attraverso le sue iterazioni. Il problema principale consiste nell'affrontare la complessità spaziale delle iterazioni iniziali. Le moderne infrastrutture cloud scalabili, come AWS e Google Cloud sono le più adatte per eseguire tali algoritmi.
- *Iterazioni divergenti*: man mano che l'algoritmo svolge le iterazioni, la quantità di dati elaborati in ogni singola iterazione aumenta. Poiché la complessità spaziale aumenta con il progredire dell'algoritmo, è importante impostare dei vincoli per evitare che il sistema diventi instabile. I vincoli possono essere impostati limitando il numero di iterazioni e/o impostando un limite alla dimensione dei dati iniziali.
- *Iterazioni piatte*: man mano che l'algoritmo svolge le iterazioni, la quantità di dati elaborati in ogni singola iterazione rimane costante. Poiché la complessità spaziale non cambia, non è necessaria particolare elasticità nelle infrastrutture.

Per calcolare la complessità spaziale, dobbiamo concentrarci su una delle iterazioni più complesse. In molti algoritmi, man mano che si converge verso la soluzione, il fabbisogno di risorse si riduce gradualmente. In questi casi, le iterazioni iniziali sono le più complesse e ci danno una stima migliore della complessità spaziale. Una volta scelto l'algoritmo, stimiamo la quantità totale di memoria da esso utilizzata, inclusa la memoria impiegate dalle strutture di dati transitorie, dall'esecuzione e dai valori di input. Questo ci darà una buona stima della complessità spaziale di un algoritmo.
Di seguito sono riportate le linee guida per ridurre al minimo la complessità spaziale.

- Quando possibile, provate a progettare un algoritmo iterativo.
- Durante la progettazione di un algoritmo iterativo, ogni volta che avete una scelta, preferite la possibilità di avere un numero maggiore di iterazioni, non un numero minore. L'idea è che un numero maggiore di iterazioni a granularità fine avrà una minore complessità spaziale.
- Gli algoritmi dovrebbero portare in memoria solo le informazioni necessarie per l'elaborazione corrente. Tutto ciò che non serve va cancellato dalla memoria.

L'analisi della complessità spaziale è un *must* per la progettazione efficiente di algoritmi. Se non viene condotta un'analisi adeguata nella progettazione di un determinato algoritmo, una disponibilità di memoria insufficiente per le strutture di dati temporanee di cui ha bisogno può innescare inutili trasferimenti dei dati su disco, che potrebbero potenzialmente influire notevolmente sulle prestazioni e sull'efficienza dell'algoritmo stesso. In questo capitolo approfondiremo la complessità temporale. La complessità spaziale sarà trattata più in dettaglio nel Capitolo 15, dove tratteremo gli algoritmi distribuiti a larga scala, i quali hanno notevoli requisiti di memoria a runtime.

Analisi della complessità temporale

Tale analisi stima quanto tempo impiegherà un algoritmo per completare il lavoro assegnato, in base alla sua struttura. A differenza della complessità spaziale, la complessità temporale non dipende dall'hardware su cui verrà eseguito l'algoritmo, ma esclusivamente dalla struttura dell'algoritmo stesso. L'obiettivo generale dell'analisi della complessità temporale è quello di cercare di rispondere a due importanti domande.

- *Questo algoritmo sarà scalabile?* Un algoritmo ben progettato dovrebbe essere pienamente in grado di sfruttare la moderna infrastruttura elastica disponibile negli ambienti di cloud computing. Un algoritmo dovrebbe essere progettato in modo tale da poter utilizzare la disponibilità di più CPU, core, GPU e memoria. Per esempio, un algoritmo utilizzato per addestrare un modello in un problema di machine learning dovrebbe essere in grado di utilizzare l'addestramento distribuito man mano che si rendono disponibili più CPU.
  Tali algoritmi dovrebbero inoltre sfruttare nuove GPU e nuovi spazi di memoria se si rendono disponibili durante l'esecuzione.
- *Quanto bene questo algoritmo gestirà dataset più grandi?*

Per rispondere a queste domande, dobbiamo determinare l'effetto sulle prestazioni di un algoritmo a mano a mano che aumentano le dimensioni dei dati, e assicurarci che l'algoritmo sia progettato in modo tale da renderlo non solo accurato, ma anche scalabile. Le prestazioni di un algoritmo stanno diventando sempre più importanti per dataset più grandi, nel mondo odierno dei "big data".
In molti casi, potremmo avere a disposizione più di un approccio per progettare l'algoritmo. L'obiettivo di condurre un'analisi della complessità temporale, in questo caso, sarà il seguente.

> Dato un determinato problema e più algoritmi per risolverlo, qual è il più efficiente in termini di efficienza temporale?

Ci possono essere due approcci di base per calcolare la complessità temporale di un algoritmo.

- *Approccio di profilazione post-implementazione*: vengono implementati più algoritmi candidati e ne vengono confrontate le prestazioni.
- *Approccio teorico pre-implementazione*: le prestazioni di ciascun algoritmo vengono approssimate matematicamente, prima ancora di eseguire l'algoritmo.

Il vantaggio dell'approccio teorico è che dipende solo dalla struttura dell'algoritmo, non dall'hardware effettivo che verrà utilizzato per eseguirlo, dalla scelta dello stack software impiegato a runtime o dal linguaggio di programmazione utilizzato per implementare l'algoritmo.

Stima delle prestazioni

Le prestazioni di un tipico algoritmo dipenderanno dal tipo di dati che gli vengono forniti come input. Per esempio, se i dati fossero già ordinati in base al contesto del problema che stiamo cercando di risolvere, l'algoritmo potrebbe funzionare in modo incredibilmente veloce. Se per eseguire il benchmark di questo particolare algoritmo venisse impiegato un input ordinato, otterremmo prestazioni irrealisticamente buone, che non rifletteranno le sue prestazioni reali, nella maggior parte degli scenari tipici. Per gestire questa dipendenza degli algoritmi dai dati di input, dobbiamo considerare diversi tipi di casi quando ci troviamo a eseguire un'analisi delle prestazioni.

Il caso migliore

Nel migliore dei casi, i dati forniti come input sono già organizzati in modo tale che l'algoritmo offra le sue prestazioni migliori. L'analisi del caso migliore fornisce il limite superiore delle sue prestazioni.

Il caso peggiore

Il secondo modo per stimare le prestazioni di un algoritmo è quello di cercare di trovare il tempo massimo possibile necessario perché porti a termine il lavoro in un determinato insieme di condizioni. Questa analisi del caso peggiore di un algoritmo è piuttosto utile, in quanto ci permette di garantire che, indipendentemente dalle condizioni, le prestazioni dell'algoritmo saranno sempre migliori di quanto emerge da questa analisi. L'analisi del caso peggiore è particolarmente utile per stimare le prestazioni quando si affrontano problemi complessi con grandi dataset. L'analisi del caso peggiore fornisce il limite inferiore delle prestazioni dell'algoritmo.

Il caso medio

Si inizia dividendo i vari possibili input in vari gruppi. Quindi, si conduce l'analisi delle prestazioni con uno degli input rappresentativi di ciascun gruppo. Infine, si calcola la media delle prestazioni di ciascuno dei gruppi.
L'analisi del caso medio non è sempre accurata, in quanto deve considerare tutte le diverse combinazioni e possibilità di input all'algoritmo, cosa non sempre facile.

Notazione Big O

La notazione Big O (è la lettera "O", non lo zero "0") fu introdotta per la prima volta da Bachmann nel 1894 in un documento di ricerca per approssimare la crescita di un algoritmo. Scrive: "…con il simbolo O(n) esprimiamo una grandezza il cui ordine rispetto a n non supera l'ordine di n" (Bachmann 1894, p. 401).
La notazione Big-O offre un modo per descrivere il tasso di crescita a lungo termine delle prestazioni di un algoritmo. In termini più semplici, ci dice come il tempo di esecuzione di un algoritmo aumenta al crescere delle dimensioni dell'input. Analizziamolo con l'aiuto di due funzioni, $f(n)$ e $g(n)$. Se diciamo $f = O(g)$, ciò che intendiamo è che quando n tende all'infinito, il rapporto $f(n) / g(n)$ rimane limitato. In altre parole, per quanto grande sia il nostro input, $f(n)$ non crescerà in modo sproporzionato più velocemente di $g(n)$.
Vediamo le funzioni:

$$f(n) = 1000n^2 + 100n + 10$$

e

$$g(n) = n^2$$

Notate che entrambe le funzioni si avvicineranno all'infinito quando n si avvicina all'infinito. Scopriamo se $f = O(g)$ applicando la definizione.
Innanzitutto, calcoliamo $f(n) / g(n)$,
che sarà uguale a $(1000n^2 + 100n + 10) / n^2 = 1000 + 100 / n + 10 / n^2$
È chiaro che $f(n) / g(n)$ è limitato e non si avvicinerà all'infinito quando n si avvicina all'infinito.
Pertanto, $f(n) = O(g) = O(n^2)$.
(n^2) rappresenta il fatto che la complessità di questa funzione aumenta con il quadrato dell'input n. Se raddoppiamo il numero di elementi di input, la complessità dovrebbe aumentare di un fattore 4.
Considerate le seguenti cinque regole quando impiegate la notazione Big-O.

Regola 1
Esaminiamo la complessità dei cicli negli algoritmi. Se un algoritmo esegue una determinata sequenza di passaggi n volte, la sua prestazione è O(n).

Regola 2
Esaminiamo i cicli annidati degli algoritmi. Se un algoritmo esegue una funzione che ha un ciclo di n passi e per ogni ciclo esegue altri n^2 passi, la prestazione totale dell'algoritmo è $O(n \times n^2)$.
Per esempio, se un algoritmo ha cicli esterni e interni aventi n passaggi, la complessità dell'algoritmo sarà rappresentata da:

$$O(n \times n) = O(n^2)$$

Regola 3
Se un algoritmo esegue una funzione $f(n)$ che richiede n passaggi e poi esegue un'altra funzione $g(n)$ che richiede n^2 passi, la prestazione totale dell'algoritmo è O($f(n) + g(n)$).

Regola 4
Se un algoritmo prende O($g(n) + h(n)$) e la funzione $g(n)$ è maggiore di $h(n)$ per n grande, le prestazioni dell'algoritmo possono essere semplificate in O($g(n)$).

Ciò significa che $O(1 + n) = O(n)$.
E $O(n + n^2) = O(n^2)$.

Regola 5
Quando si calcola la complessità di un algoritmo, si possono ignorare i multipli costanti. Se k è una costante, $O(kf(n))$ è uguale a $O(f(n))$.
Inoltre, $O(f(k \times n))$ è uguale a $O(f(n))$.
Quindi $O(5n^2) = O(n^2)$.
E $O((3n^2)) = O(n^2)$.
Alcune considerazioni.

- La complessità quantificata dalla notazione Big O è solo una stima.
- Per dati di piccole dimensioni inferiori, non è importante la complessità temporale. n^0 nel grafico definisce la soglia minima *sopra* la quale ci interessa trovare la complessità.
- La complessità temporale $T(n)$ è maggiore della funzione originale. Una buona scelta di $T(n)$ proverà a creare un limite superiore stretto per $f(n)$.

La Tabella 1.1 riassume i diversi tipi di notazione Big O trattati in questo paragrafo.

Tabella 1.1 Tipi di notazione Big O.

Classe di complessità	Nome	Esempi di operazioni
$O(1)$	Costante	Aggiungere, estrarre o modificare un elemento.
$O(\log n)$	Logaritmica	Trovare un elemento in un array ordinato.
$O(n)$	Lineare	Copiare, inserire, cancellare, iterare.
$O(n^2)$	Quadratica	Cicli annidati.

Complessità costante, O(1)

Se un algoritmo impiega la stessa quantità di tempo per essere eseguito, indipendentemente dalla dimensione dei dati di input, si dice che viene eseguito in *tempo costante*, rappresentato da O(1). Prendiamo l'esempio dell'accesso all'n-esimo elemento di un array. Indipendentemente dalle dimensioni dell'array, ci vorrà un tempo costante per ottenere il risultato. Per esempio, la seguente funzione restituisce il primo elemento dell'array e ha una complessità pari a O(1):

```
def get_first(my_list):
    return my_list[0]

get_first([1, 2, 3])
1

get_first([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
1
```

Considerazioni.

- L'aggiunta di un nuovo elemento in uno stack avviene utilizzando un'operazione di `push` e la rimozione di un elemento da uno stack avviene utilizzando un'operazione

di `pop`. Indipendentemente dalle dimensioni dello stack, ci vorrà lo stesso tempo per aggiungere o rimuovere un elemento.

- Nell'accesso a un elemento di una tabella hash, notate che tale struttura memorizza i dati in modo associativo, usando coppie chiave-valore.

Complessità lineare, O(n)

Si dice che un algoritmo ha una *complessità lineare*, rappresentata da O(*n*), quando il tempo di esecuzione è direttamente proporzionale alla dimensione dell'input. Un semplice esempio consiste nell'aggiungere elementi a una struttura di dati unidimensionale:

```
def get_sum(my_list):
    sum = 0
    for item in my_list:
        sum = sum + item
    return sum
```

Notate il ciclo principale dell'algoritmo. Il numero di iterazioni del ciclo aumenta linearmente al crescere di *n*, producendo una complessità O(*n*):

```
get_sum([1, 2, 3])
6

get_sum([1, 2, 3, 4])
10
```

Alcuni altri esempi di operazioni di questo tipo sono i seguenti:

- ricerca di un elemento;
- ricerca del valore minimo fra tutti gli elementi di un array.

Complessità quadratica, O(n^2)

Si dice che un algoritmo viene eseguito in un tempo quadratico se il tempo di esecuzione di un algoritmo è proporzionale al quadrato della dimensione dell'input; un esempio è una semplice funzione che somma gli elementi di un array bidimensionale, come la seguente:

```
def get_sum(my_list):
    sum = 0
    for row in my_list:
        for item in row:
            sum += item
    return sum
```

Notate il ciclo interno, annidato nel ciclo principale. Questo ciclo annidato conferisce al codice precedente la complessità O(n^2):

```
get_sum([[1, 2], [3, 4]])
10
```

```
get_sum([[1, 2, 3], [4, 5, 6]])
21
```

Un altro esempio è l'algoritmo *bubble sort* (trattato nel Capitolo 2).

Complessità logaritmica, O(logn)

Si dice che un algoritmo viene eseguito in un tempo logaritmico quando il tempo di esecuzione dell'algoritmo è proporzionale al logaritmo della dimensione dell'input. A ogni iterazione, la dimensione dell'input diminuisce di un fattore multiplo costante. Un esempio di complessità logaritmica è la ricerca binaria. L'algoritmo di ricerca binaria trova un determinato elemento in una struttura di dati unidimensionale, come una lista Python. Gli elementi all'interno della struttura di dati devono essere ordinati in ordine decrescente. L'algoritmo di ricerca binaria è implementato tramite una funzione denominata `search_binary`, come segue:

```
def search_binary(my_list, item):
    first = 0
    last = len(my_list) - 1
    found_flag = False
    while(first <= last and not found_flag):
        mid = (first + last) // 2
        if my_list[mid] == item :
            found_flag = True
        else:
            if item < my_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
        return found_flag

searchBinary([8,9,10,100,1000,2000,3000], 10)
True

searchBinary([8,9,10,100,1000,2000,3000], 5)
False
```

Il ciclo principale sfrutta il fatto che la lista è ordinata. Divide la lista a metà a ogni iterazione, fino ad arrivare al risultato.
Dopo aver definito la funzione, il controllo da cui dipende la ricerca di un determinato elemento si trova nelle righe 11 e 12. L'algoritmo di ricerca binaria è ulteriormente trattato nel Capitolo 3.
Notate che fra i quattro tipi di notazione Big O presentati, $O(n^2)$ offre le prestazioni peggiori e O(log*n*) offre le prestazioni migliori. In effetti, le prestazioni O(log*n*) possono essere considerate il *gold standard* per le prestazioni di qualsiasi algoritmo (cosa che però non sempre viene raggiunta). D'altra parte, $O(n^2)$ non è così male come $O(n^3)$, ma gli algoritmi che rientrano in questa classe non possono essere utilizzati sui big data, poiché la loro complessità temporale impone dei limiti alla quantità di dati che possono elaborare,

realisticamente. Le prestazioni dei quattro tipi di notazioni Big O sono rappresentate nella Figura 1.3.

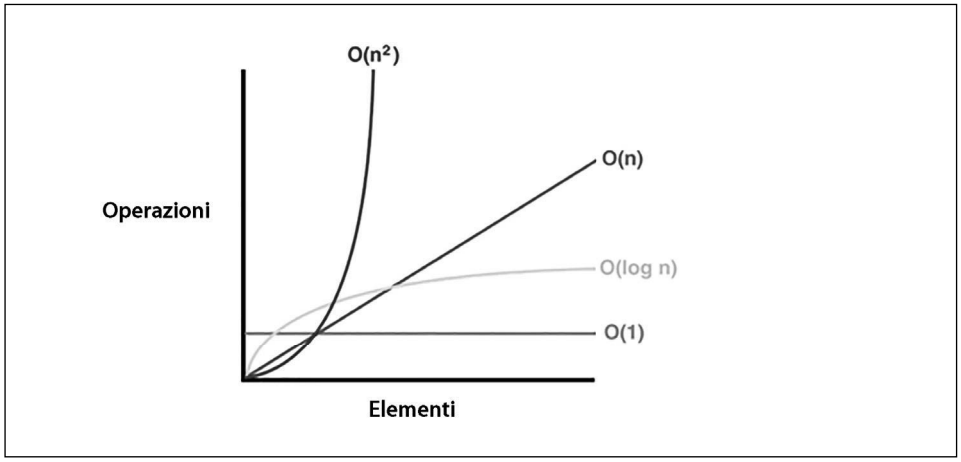

Figura 1.3 Grafico delle complessità Big O.

Un modo per ridurre la complessità di un algoritmo consiste nello scendere a compromessi sulla sua accuratezza, producendo un tipo di algoritmo chiamato *algoritmo approssimato*.

Selezione di un algoritmo

Come sapere quale è la soluzione migliore? Quale algoritmo è più veloce? Analizzando la complessità temporale di un algoritmo è possibile rispondere a questo tipo di domande. Facciamo un semplice esempio in cui l'obiettivo è ordinare una lista di numeri. Ci sono molti algoritmi prontamente disponibili per svolgere questo lavoro. Il problema è come scegliere quello giusto.

Innanzitutto, un'osservazione che si può fare è che la lista non contiene troppi numeri, quindi non ha importanza quale algoritmo scegliamo per ordinarla. Pertanto, se la lista contiene solo 10 numeri ($n = 10$), non importa quale algoritmo scegliamo, poiché l'intera operazione non richiederà più di pochi microsecondi, anche con un algoritmo molto inefficiente. Ma all'aumentare di n, la scelta dell'algoritmo "giusto" inizia a fare la differenza. Un algoritmo mal progettato potrebbe richiedere un paio d'ore, mentre un algoritmo ben progettato potrebbe completare l'ordinamento della lista in un paio di secondi. Pertanto, per dataset di input più grande, è davvero sensato investire tempo e impegno, eseguire un'analisi delle prestazioni e scegliere l'algoritmo che svolgerà il lavoro richiesto nel modo più efficiente.

Convalida di un algoritmo

La convalida di un algoritmo conferma che sta effettivamente fornendo una soluzione matematica al problema che stiamo cercando di risolvere. Un processo di convalida do-

vrebbe controllare i risultati per quanti più valori di input possibili e anche per quanti più tipi di valori di input possibili.

Algoritmi esatti, approssimati e randomizzati

La convalida di un algoritmo dipende anche dal tipo di algoritmo, in quanto le tecniche di test sono differenti. Differenziamo prima di tutto gli algoritmi deterministici da quelli randomizzati.

Per gli *algoritmi deterministici*, un determinato input genera sempre esattamente lo stesso output. Ma per alcune classi di algoritmi, come input può essere presa una sequenza di numeri casuali, il che rende l'output differente a ogni esecuzione dell'algoritmo. L'algoritmo di clustering k-means (Figura 1.8), descritto in dettaglio nel Capitolo 6, è un esempio di questo tipo.

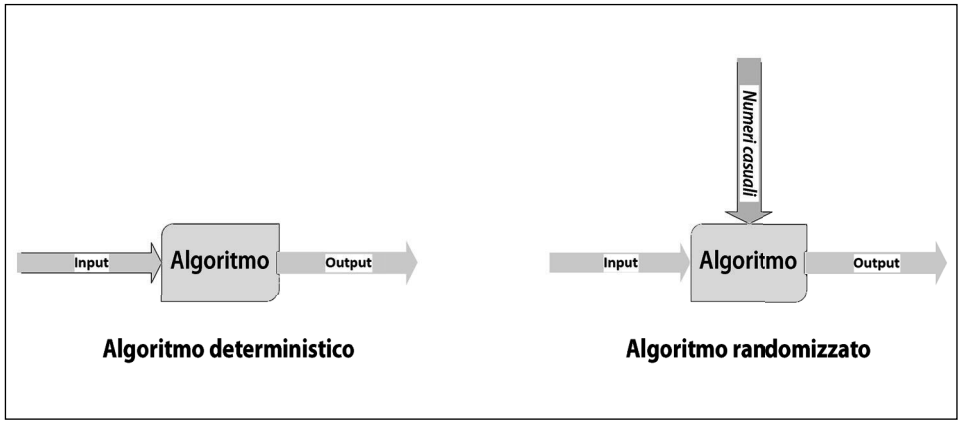

Figura 1.4 Algoritmi deterministici e randomizzati.

Gli algoritmi possono anche essere suddivisi nei seguenti due tipi, sulla base delle ipotesi o delle approssimazioni utilizzate per semplificare la logica con lo scopo di renderli più veloci.

- *Algoritmi esatti*: ci si aspetta che gli algoritmi esatti producano una soluzione precisa senza richiedere alcuna ipotesi o approssimazione.
- *Algoritmi approssimati*: quando la complessità del problema è eccessiva per le risorse disponibili, semplifichiamo il problema sulla base di alcune ipotesi. Gli algoritmi approssimati, basati su queste semplificazioni o ipotesi, non ci offrono una soluzione precisa.

Facciamo un esempio per capire la differenza fra un algoritmo esatto e uno approssimato: il famoso problema del commesso viaggiatore, presentato nel 1930. Un commesso viaggiatore vi sfida a trovare il percorso più breve per un determinato venditore che deve visitare tutte le città di un elenco e poi tornare alla base. Il primo tentativo di fornire la soluzione prevedrà la generazione di tutte le possibili permutazioni delle città e la scelta della combinazione di città più economica. È ovvio che la complessità temporale inizia a diventare ingestibile oltre le 30 città.

Se il numero di città è superiore a 30, un modo per ridurre la complessità consiste nell'introdurre alcune approssimazioni e ipotesi.
Per gli algoritmi approssimati, è importante impostare le aspettative di accuratezza nella raccolta dei requisiti. La convalida di un algoritmo approssimato richiede la verifica che l'errore dei risultati rientri in un intervallo accettabile.

Spiegabilità

Quando gli algoritmi vengono utilizzati per situazioni critiche, diventa importante avere la capacità di spiegare il motivo che è alla base di ogni risultato ogni volta che sia necessario. Ciò permette di assicurarci che le decisioni basate sui risultati degli algoritmi non introducano distorsioni.
La capacità di identificare esattamente le caratteristiche che vengono utilizzate, direttamente o indirettamente, per prendere una decisione è chiamata *spiegabilità di un algoritmo*. Gli algoritmi, quando vengono applicati a casi d'uso critici, devono essere valutati in termini di bias e pregiudizi. L'analisi etica degli algoritmi è diventata un elemento standard del processo di convalida cui occorre sottoporre quegli algoritmi che possono influenzare processi decisionali che riguardano la vita delle persone.
Per gli algoritmi che si occupano di deep learning, la spiegabilità può essere difficile da raggiungere. Per esempio, se un algoritmo rifiuta la richiesta di un mutuo da parte di una persona, è importante pretendere trasparenza e avere la capacità di spiegarne il motivo.
La spiegabilità degli algoritmi è un'area attiva di ricerca. Una delle tecniche più efficaci, sviluppata recentemente, si chiama *Local Interpretable Model-agnostic Explanation* (LIME), proposta negli atti della 22ª *Association for Computing Machinery* (ACM) alla conferenza internazionale *Special Interest Group on Knowledge Discovery and Data Mining* (SIGKDD) sulle scoperte della conoscenza e il data mining, nel 2016. La tecnica LIME si basa su un concetto: vengono introdotti piccoli cambiamenti all'input per ogni istanza e quindi viene fatto il tentativo di mappare il confine decisionale locale per quell'istanza. In tal modo è in grado di quantificare l'influenza di ciascuna variabile di tale istanza.

Riepilogo

Questo capitolo ha trattato le basi degli algoritmi. Innanzitutto, abbiamo appreso le diverse fasi dello sviluppo di un algoritmo. Abbiamo trattato i diversi modi di specificare la logica di un algoritmo che sono necessari per progettarlo. Quindi, abbiamo esaminato il modo in cui si progetta un algoritmo. Abbiamo poi imparato due diversi modi per analizzare le prestazioni di un algoritmo. Infine, abbiamo studiato i diversi aspetti della convalida di un algoritmo.
Con lo studio di questo capitolo, dovreste essere in grado di comprendere lo pseudocodice di un algoritmo e le diverse fasi dello sviluppo e dell'implementazione di un algoritmo. Avete anche imparato a usare la notazione Big O per valutare le prestazioni di un algoritmo.
Il prossimo capitolo riguarda le strutture di dati utilizzate negli algoritmi. Inizieremo esaminando le strutture di dati disponibili in Python. Vedremo quindi come possiamo utilizzare queste strutture di base per creare strutture di dati più sofisticate, come gli stack, le code e gli alberi, che sono necessarie per sviluppare algoritmi complessi.