Algoritmi a confronto: Perceptron e Decision Tree

Chiara Albisani

10 Febbraio 2020

1 Introduzione

In questa relazione si vogliono confrontare le prestazioni, in termini di curve di apprendimento, degli algoritmi Perceptron e Decision Tree, su un dataset di articoli di moda, reso disponibile da Zalando. Lo scenario di lavoro è quello dell'apprendimento supervisionato, nello specifico della classificazione, in cui l'obiettivo è riuscire a determinare la classe di appartenenza delle immagini sotto esame.

2 Analisi teorica

2.1 Perceptron

Il Perceptron è un classificatore lineare, ovvero che sfrutta una funzione lineare nella forma $f(x) = wx^T + b$ per discriminare tra due o più classi di esempi, come nel caso delle nostre 10 classi di immagini.
La sua versione binaria si presta ad una rappresentazione geometrica molto efficace che permette di intuirne il modus operandi. Infatti l'idea sfruttata dall'algoritmo è quella che lo spazio degli esempi di training $X = (x^{(1)}, x^{(2)}, \cdots, x^{(n)})$ sia diviso in due parti dall'iperpiano definito dall'equazione $w^T x + b = 0$, dove il vettore w 'weight', definisce la direzione ortogonale all'iperpiano, e quindi il suo orientamento nello spazio, mentre b 'bias', variando, descrive di quanto l'iperpiano deve traslare lungo la direzione descritta da w. La classificazione di ogni esempio $(x^{(i)}, y^{(i)})$ avviene moltiplicando $y^{(i)}$ per l'equazione dell'iperpiano e andando a verificare se tale prodotto $y^{(i)}(w^T x^{(i)} + b)$ è maggiore o minore di zero. Nel caso i cui il segno del prodotto sia positivo, significa che l'esempio in questione è stato correttamente classificato, cioè si trova dalla parte di piano 'corretta', seguendo l'interpretazione geometrica, nel caso contrario l'esempio non è classificato correttamente e quindi l'iperpiano viene traslato, aggiornando opportunamente i parametri di controllo w e b. L'approccio utilizzato è di tipo greedy, in quanto ogni volta che si riscontra un errore di classificazione, questo viene corretto immediatamente. L'algoritmo termina (converge) se esiste un iperpiano che classifica correttamente i dati di training, ovvero se i dati sono linearmente separabili, e w e b di tale iperpiano possono essere quindi ritornati. Nel caso in esame l'algoritmo dovrà trattare 10 iperpiani, uno per ogni classe, combinando le operazioni sopra descritte.

2.2 Decision Tree

Il Decision Tree è un metodo di apprendimento supervisionato, in cui, nell'ambito della classificazione, il percorso verso una possibile classe è codificato in un particolare cammino dell'albero. In questa struttura i nodi foglia rappresentano le classificazioni, i nodi interni, radice compresa,

costituiscono le cosiddette condizioni di split, mentre gli archi descrivono le conseguenze della scelta di una particolare condizione. Il termine condizione di split è particolarmente indicativo del comportamento dell'algoritmo. Infatti l'idea è quella di partizionare lo spazio degli esempi, assecondando una decisione piuttosto che un'altra, con l'obiettivo di ridurlo progressivamente in sottoinsiemi sempre più piccoli e omogenei circa le categorie rispetto alle quali si deve classificare. La possibilità di fornire, oltre alla classificazione, anche una spiegazione del modo in cui si è arrivati a tale scelta, sfruttando la visualizzazione della struttura ad albero, costituisce uno degli aspetti più interessanti e che motiva l'utilizzo del Decision Tree. La complessità di tale classificatore ha ovviamente a che fare con il numero di nodi gestito dall'albero. Un albero caratterizzato da un alto numero di nodi, e quindi da molte possibilità di partizionamento, è un modello troppo complesso e non funzionale, in quanto l'essere molto fine nella suddivisione degli esempi porta l'algoritmo a lavorare perfettamente sui dati di training ma ad avere difficoltà sui dati di testing e quindi sulla generalizzazione. Questo tipo di comportamento è definito $overfitting$. Determinare un Decision Tree ottimo, che quindi classifica correttamente utilizzando il numero minore di nodi, è un problema NP-completo, ma nella pratica esistono delle euristiche attraverso le quali si possono fare delle scelte ottime ad ogni nodo, riducendo il numero complessivo di nodi su cui lavorare.

3 Documentazione del codice

In questa sezione sono presentati i contenuti dei file.py utilizzati per l'analisi sperimentale.

3.1 main.py

All'interno della funzione **main** è implementata la logica di testing.
Prima di tutto si carica il dataset, scomposto per costruzione in dati di training (60.000 immagini grayscale di $28 * 28$ pixel, 60.000 labels i cui valori variano tra 0 e 9) e dati di testing (10.000 immagini grayscale di $28 * 28$ pixel, 10.000 labels, i cui valori variano tra 0 e 9). In particolare la chiamata alla funzione **loadDataset** verifica la presenza dei dati in questione e in caso di assenza (come al momento della prima esecuzione del codice) provvede a reperirli direttamente dal repository di github e a salvarli nella cartella ./data. Queste operazioni sono svolte combinando l'utilizzo dei pacchetti os, shutil, tempfile e soprattutto GitPython, che permette di clonare da un repository github le cartelle e/o i file di cui si ha bisogno. Al caricamento segue la fase di pre processing dei dati. Questa fase, sintetizzata nella chiamata alla funzione **scaleData**, è molto importante in quanto, avere dati che assumono valori molto distanti tra di loro, può portare ad errori e appesantire l'esecuzione. Per normalizzare le immagini di training e di testing si è optato per l'utilizzo di *StandardScaler*, disponibile nella libreria scikit-learn, che trasforma ogni immagine in un vettore di $28 * 28 = 784$ valori a media nulla e varianza zero. Un 'altra possibile normalizzazione poteva essere: convertire i vettori delle immagini di training e di testing da interi a float e poi dividerli per 255.0. Sapendo infatti che i pixel di un immagine grayscale variano tra 0 e 255, dividere per 255.0 significa portare i valori dei vettori a variare in un range compreso tra 0 e 1. A questo punto si dichiarano le istanze dei due modelli, chiamando i corrispondenti metodi della libreria scikit-learn. Si è scelto di lasciare la lista dei parametri di default, previsti dalla libreria, inalterata, per entrambi i modelli, anche se una modulazione di alcuni parametri potrebbe essere opportuna e necessaria, specialmente in caso di limitate risorse di memoria. Per quanto riguarda il Decisione Tree, a questo proposito, parametri molto interessanti sono max_depth, la massima profondità dell'albero, $min\_samples\_split$, il minimo numero di campioni richiesto per fare fare split di un nodo interno, e $min\_impurity\_decrease$, la soglia di impurità sotto la quale ad un nodo non viene più applicato lo split.
Per il Perceptron, tra i parametri più interessanti ci sono max_iter e $eta0$, che corrispondono rispettivamente al numero di epoche, ovvero il numero di volte in cui si ripete il ciclo di controllo

sui campioni del dataset per vedere se risultano classificati correttamente oppure no, e il tasso di apprendimento (learning rate), che consiste nel fattore costante che si moltiplica in fase di aggiornamento dei parametri di controllo w e b.
Individuati i parametri di interesse e soprattutto i loro possibili range di valori, si potrebbe pensare di operare una ricerca al fine di determinare i valori migliori per questi parametri, ed allenare i modelli con quest'ultimi. L'idea potrebbe essere quella di costruire una funzione che in base al modello e alla lista di parametri passata applica *GridSearchCV* della libreria scikit-learn, la quale esercita ciascun modello con tutte le combinazioni di parametri possibili, salvando i corrispettivi valori di accuratezza (se l'accuratezza è la metrica scelta), e ritorna il modello con i parametri migliori. Dato che *GridSearchCV* potrebbe essere molto pesante, si può sostituire con *RandomizedSearchCV*, semplicemente aggiungendo tra i parametri *n_iter*, per fissare il numero massimo di combinazioni dei parametri che si vuole testare.
Infine si disegnano le curve di apprendimento per entrambi i modelli, con la chiamata alla funzione **plotLearningCurve**.

3.2 functions.py

In questo file sono presenti tutte le funzioni di supporto all'esecuzione del main. Oltre alle già citate **loadDataset** e **scaleData**, sono presenti due funzioni, **showImage** e **plotImages**, per visualizzare le immagini del dataset, insieme alla funzione **plotLearningCurve**, su cui è bene fare delle precisazioni.

- **showImage**: Sfruttando le funzioni della libreria matplotlib, stampa l'immagine desiderata con annessa una label. Nel caso in cui non sia specificato il vettore delle predizioni come parametro (che di default è None), la label è proprio quella che corrisponde all'immagine selezionata, altrimenti si stampano sia la label predetta che la label corrispondente all'immagine, utilizzando il colore rosso o blu per indicare se la predizione è sbagliata o corretta.

- **plotImages**: La funzione permette di stampare in una sola figura un certo numero di immagini, specificato sotto forma di numero di righe per numero di colonne, utilizzando la funzione **showImage**.

- **plottingLearningCurve**: La funzione prevede un ciclo esterno sul numero di iterazioni passato in ingresso, all'interno del quale si 'disordinano' gli indici di training e di testing e poi con un ciclo interno sulle dimensioni variabili del training set, si esercita il modello, si calcola la sua accuratezza, con la funzione *accuracy_score* della libreria scikit-learn, rispettivamente sui dati di training e di testing, salvando i risultati nelle matrici (vettore di vettori) $scoresTr$ e $scoresTe$, di dimensioni $iter * len(trainSizes)$. Successivamente si calcolano le medie e le deviazioni standard per ogni colonna delle matrici, ottenendo i vettori $trainMeanScores$, $trainStdScores$, $testMeanScores$ e $testStdScores$, delle stesse dimensioni di $trainSizes$. A questo punto si può disegnare, sullo stesso grafico, l'andamento del training e del testing score al variare della dimensione del training set.

4 Analisi sperimentale

Gli esperimenti sono stati eseguiti su un dispositivo Lenovo Y50-70, processore Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz, basato su x64, RAM installata 8.00 GB, con sistema operativo a 64 bit, Windows 10 Home.
I risultati ottenuti sono visibili in Figure 1.
Come era lecito aspettarsi per pochi dati di training entrambi i classificatori manifestano un ottimo comportamento in termini di training score (pari a 1) e un pessimo comportamento

in termini di testing score, sintomo evidente di overfitting. Si dice infatti in questi casi che l'algoritmo apprende il rumore dei dati e non il segnale, intendendo con questo che l'algoritmo non riesce ad intuire la distribuzione dei campioni e quindi a generalizzare bene. D'altra parte l'ampio gap tra le due curve, indice di un alto grado di varianza, racconta che ad incidere sulle prestazioni dell'algoritmo sui cosiddetti 'out-of-samples' è la particolare partizione di dati utilizzata per l'apprendimento. Questo è il motivo per cui si è scelto di ripetere i calcoli degli score più volte, cambiando randomicamente gli indici del training e del testing set, per poi considerarne i valori medi per ogni dimensione del training set, in modo da avere una percezione del comportamento quanto più realistica.
Con l'aumentare dei dati di apprendimento il training score si riduce mentre il testing score aumenta, riducendo il gap tra le due curve e quindi la varianza. In particolare nel grafico del Perceptron le due curve raggiungono una buona convergenza, intorno a 0.8, mentre la distanza tra le due curve del Decision Tree si mantiene ancora piuttosto lontana dalla convergenza. In quest'ultimo caso, probabilmente aumentare i dati di training, se questo fosse possibile, porterebbe a migliorare le prestazioni per quanto riguarda il testing score. Il fatto che il training score del Decision Tree si mantenga costante e pari ad 1 al variare della dimensione del training set, è un comportamento inatteso, ma che si può forse spiegare con la non modulazione degli iperparametri del modello. In una situazione del genere, il Decision Tree ha piena possibilità di espansione, sia in termini di profondità, sia in termini di numero di foglie, e questo porta alla costruzione di un classificatore che modella perfettamente i dati di apprendimento. La piena capacità di espansione oltre ad essere incisiva dal punto di vista dello spazio occupato, implica un tempo di esecuzione molto alto, anche nel caso di un numero non elevato di iterazioni.

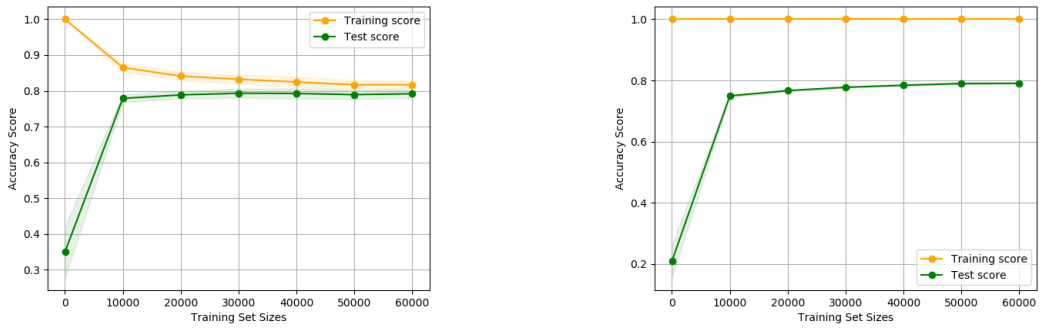

Figure 1: Curve di apprendimento del Perceptron (a sinistra) e del Decision Tree (a destra)