Random Forest

Gian Maria Gennai

Progetto per Intelligenza Artificiale.

Ingegneria Informatica
Università degli studi di Firenze

Indice

1 Introduzione

All'interno di questa relazione andremo a studiare l'algoritmo **Random Forest** comparando una versione implementata da me e una implementata con *scikit-learn*.
In generale l'algoritmo Random Forest è molto semplice, flessibile e preciso, per questo è utilizzato in molti ambiti. L'algoritmo mette insieme un certo numero N di singoli alberi decisionali per avere un risultato finale più robusto rispetto a quello ottenuto da un singolo albero.

2 Progettazione

Per lo sviluppo di questo progetto ho utilizzato esclusivamente il linguaggio **Python** sull'IDE di sviluppo fornito da *IntelliJ*, PyCharm.
Prima di iniziare scrivere codice, mi sono documentato su come andare ad addestrare un singolo albero e poi anche come metterne tanti insieme per creare una foresta, sia leggendo spiegazioni trovate da più fonti e sia andando a rileggere lo pseudocodice dal libro *Artificial Intelligence* di *Russel & Norvig*.
Per la parte riguardante i dati che ho utilizzato per allenare e testare la mia rete ho invece scaricato ed utilizzato alcuni dataset da Machine Learning Repository. Ho testato il codice su tre dataset differenti e questi sono Iris, Blood Transfusion e Fertility.
L'implementazione dell'Albero singolo mi ha portato via più tempo perché è quella che effettivamente fa il lavoro '*sporco*', invece il mettere insieme tutti gli alberi e controllare che i risultati finali producessero qualcosa di coerente è stato più rapido.

Adesso andremo a vedere nel dettaglio cosa è stato elaborato.

3 Implementazione

All'interno del progetto troviamo cinque file .py che mi vanno a creare la struttura generale.
Il file *main.py* è quello con cui possiamo interagire caricando il dataset che più ci piace e cambiando alcuni parametri. In particolare possiamo andare a caricare i vari dataset grazie al comando *pd.read_csv(...)*, (*pd* è un abbreviazione impostata da me per richiamare **pandas**, una libreria di Python, che ho utilizzato per manipolare i dataset scaricati). Dopo aver caricato i vari dataset e aver manipolato i due valori che possiamo modificare, percentage e nEstimator, possiamo cliccare su RUN e aspettare i risultati. La variabile percentage indica la percentuale di dataset da dedicare al test, nEstimator verrà spiegato in seguito.
Appena premuto RUN entreremo all'interno del file *RandomForest.py* tramite il metodo *randomForest()*. All'interno di questo file, come possiamo ben capire dal nome, viene creata la foresta e per fare questo abbiamo un ciclo for che ha un numero di ripetizioni impostato da noi nel main.py (tramite l'attributo

nEstimator) il quale tutte le volte mi allena un albero differente.
Tralasciando tutte le parti di codice superflue dobbiamo sottolineare due cose importanti, la prima è che noi tutte le volte che creiamo un nuovo albero chiamiamo il metodo *boostrap* da *Utility.py* che mi permette di prendere il dataset iniziale e crearne uno nuovo delle stesse dimensioni ma con righe prese in ordine casuale da quello originale(possiamo avere più righe uguali), la seconda è che noi andiamo a prendere un sottoinsieme di tutti gli attributi che abbiamo a disposizione e non l'insieme totale e questi due punti mi fanno si di avere molta più differenza nella costruzione di un albero rispetto ad un altro in modo da poter coprire più casi possibili.
Fatte queste procedure entriamo nel vivo della creazione dell'albero quindi all'interno del file *Tree.py*. Qui, in questo file, abbiamo più metodi che si chiamano in automatico ad ogni passo, il metodo *decisionTree()* è il cuore del processo e l'albero è costruito sulla ricorsione di quest'ultimo metodo. Tutta la struttura che si crea è salvata all'interno di un **Dizionario** che è un elemento fornito dal linguaggio e formato da una chiave e un valore.
Ogni riga del dataset è composta da vari attributi, ogni riga fa poi parte di una classe. Dobbiamo immaginare l'albero che stiamo creando come una struttura dove ogni nodo è un certo valore dell'attributo appartenente al dataset che stiamo utilizzando, ogni valore che assume un certo attributo è contenuto all'interno delle righe del dataset e l'insieme dei valori degli attributi che appartengono ad una riga della tabella mi formano un percorso, quando il percorso arriva alla fine abbiamo una foglia che prenderà il valore non più di un attributo ma di una classe.
Avendo ora spiegato come è formato un albero è facile capire che quando poi io vado a passare i dati di test, seguendo il valore degli attributi che ho in una riga di test io possa arrivare ad assegnargli una classe. Se il numero di dati disponibile per l'apprendimento è molto piccolo c'è il rischio che la rete riesca a non coprire tutti i casi possibili e quindi può capitare che arrivi un test e non trovi la sua classe di appartenenza, in quel caso gli viene assegnato il valore **None** ovvero nullo. Ogni volta che viene analizzato un nuovo sottoinsieme del dataset iniziale viene scelto l'attributo che riesce a darmi performance migliori e per fare questo viene calcolato il **gain** come:

$$gain = Cost(D) - \sum \frac{\mathrm{mod} D_i}{\mathrm{mod} D} * Cost(D_i)$$

la funzione Cost(), ovvero la funzione che stima il costo, che vediamo all'interno della formula è quella che può essere implementata in diversi modi:

- Errore di classificazione:
$$1 - P_i$$
- Gini:
$$\sum P_i(1 - P_i)$$
- Entropia:
$$-\sum P_i \ln P_i$$

Io ho utilizzato l'entropia perché è quella che rende valori migliori, l'errore di classificazione non è consigliato utilizzarlo e invece l'indice Gini e l'entropia si assomigliano come comportamento ma quest'ultima è leggermente migliore. Sempre per quanto riguarda la formula del gain il valore D rappresenta il Dominio totale e Di rappresenta il dominio specifico di una determinata classe di appartenenza.
Detto questo viene calcolato il massimo dei gain di tutti gli attributi e preso l'attributo con il gain più alto.
Andando poi a mettere i valori nel dizionario appaiono alcuni attributi che sono **notPure** cioè ci sono righe con quell'attributo che appartengono a classi di appartenenza differenti e quindi dove si presenta questa stringa noi andiamo a iterare ricorsivamente su quel valore sperando di arrivare ad avere una classe pura finale e quindi una foglia prima che finiscano gli attributi, senno come già spiegato avremo il valore *None*.
Creato l'albero torniamo al file RandomForest.py, lo salviamo in una lista e procediamo con passare ogni riga della tabella di test per poi salvare le predizioni ottenute.
Fatto questo processo per tutti gli alberi della foresta, andiamo a vedere i risultati ottenuti che adesso li abbiamo tutti insieme dentro un unica lista. Ogni riga della tabella dei test avrà un numero di predizioni, giuste o sbagliate, salvate e in numero pari al numero di alberi che sono stati creati.
A questo punto controlliamo il valore della classe ripetuta più volte per ogni riga di test e scegliamo quella classe con il valore più alto. Questo è il metodo della Classificazione.
Adesso viene restituito nel main il livello di accuratezza con il quale sono stati predetti i vari valori.
Fatto questo, chiamando il metodo *sklearnRandomForest()* del file *SklearnRF.py*, anche qui, viene restituito il valore di precisione della predizione. Le funzioni chiamate all'interno del file SklearnRF.py non sono spiegate, per ulteriori informazioni controllare sul sito scikit-learn.

4 Dati sperimentali

Adesso andiamo ad analizzare i dati che ci vengono restituiti dai due metodi chiamati nel main.
Come è stato spiegato sopra, i valori restituiti dai metodi *randomForest(...)* e *sklearnRandomForest(...)*, indicano l'accuratezza con cui è stata fatta la predizione. Entrambi restituiscono un valore compreso tra 0 e 1.
Questi metodi per il calcolo dell'accuratezza vanno semplicemente a contare quanti elementi della tabella di test sono stati etichettati in modo corretto e dopo vanno a dividere quel valore per il numero di righe totali della tabella.
I valori che ho testato fanno riferimento ai tre dataset citati sopra, in più ho fatto fare la prova con 10,50 e 100 alberi nella foresta.
Possiamo vedere nelle tabelle sotto i valori restituiti dalla predizione fatta sulla mia foresta rispetto a quella fatta sulla foresta implementata da scikit-learn. La

comparazione fra i due va fatta con attenzione perché delle volte capita che i dati di train passati siano migliori da una parete rispetto che dall'altra e viceversa(i dati sono sempre gli stessi ma le configurazioni con cui li prendiamo cambiano).

My Random Forest			
N° Trees	Iris	Blood Transfusion	Fertility
10	0.66	0.7	0.7
50	0.73	0.70	0.7
100	0.6	0.6	0.8

SCIKIT-LEARN Random Forest			
N° Trees	Iris	Blood Transfusion	Fertility
10	0.87	0.74	0.8
50	1	0.76	0.9
100	0.9	0.69	0.8

Per concludere posso aggiungere che il tempo che impiega l'algoritmo implementato dalla libreria è molto più veloce rispetto al mio e questo è dato dai numerosi cicli che mi sono ritrovato a dover mettere all'interno della struttura, di fatti testando il codice con dataset molto grandi o andando a aumentare troppo il numero di alberi nella foresta, il tempo di risposta cresceva molto.