Prediction accuracy degli alberi di decisione utilizzando il rule post-pruning

Daniele Ambrosino

Febbraio 2020

1 Introduzione

Gli alberi di decisione rappresentano uno dei più semplici ed efficaci modelli predittivi basati sul machine learning. Tuttavia, uno dei principali svantaggi è la loro sensibilità al training set utilizzato, fattore che spesso porta al fenomeno dell'overfitting. Ciò si verifica quando il modello si adatta a caratteristiche che sono specifiche solo del training set, ma che non hanno riscontro nel resto dei casi. Questo accade quando si utilizza un training set troppo scarno, o al contrario si effettua un training troppo prolungato. L'overfitting porta la capacità predittiva dell'albero di decisione sullo stesso training set ad aumentare, mentre le prestazioni sui dati non visionati peggiorano.

Inoltre, durante il processo di costruzione dell'albero, a causa di esempi "rumorosi" o in generale poco rilevanti, si possono generare dei cammini non utili ai fini della predizione, che anzi aumentano soltanto la complessità computazionale senza apportare alcun beneficio alla bontà del modello.

L'obiettivo di questo elaborato è fornire un'implementazione conveniente di tale modello di machine learning, ed analizzare l'impatto della tecnica del *rule post-pruning* sull'accuratezza predittiva degli alberi di decisione.

2 Implementazione

2.1 Contesto

L'intero codice è stato scritto in linguaggio Python. L'algoritmo e le strutture dati in questione sono stati progettati per apprendere e predire adeguatamente utilizzando le informazioni del dataset Adult dell'UCI Machine Learning Repository.
Tale dataset contiene attributi sia discreti che numerici: per questo motivo l'algoritmo deve poter generare un albero di decisione che sappia gestire entrambi i tipi di dato. Il dataset conteneva esempi con attributi mancanti, che per semplicità sono stati rimossi.

2.2 Scelte implementative

Il repository forniva il dataset suddiviso in due file: `adult.data` ($\approx$ 30'000 entries) ed `adult.test` ($\approx$ 15'000 entries) Ho suddiviso `adult.data` in due file:

`adult.training` ($\approx$ 20'000 entries) e `adult.validation` ($\approx$ 10'000 entries), da utilizzare rispettivamente per il training e la validation.

La classe `DecisionTreeLearner` comprende tutti i metodi per la generazione dell'albero, il pruning e la visita. Il metodo `build_tree()` avvia la costruzione dell'albero di decisione, ed è un'implementazione dell'algoritmo DECISION-TREE-LEARNING descritto in R/N §18.3.

Nel codice, la funzione IMPORTANCE è stata rinominata `get_best_node()`. Tale metodo analizza gli esempi del training set forniti, e sceglie tra gli attributi disponibili quello che divide nel modo più efficace gli esempi nella prospettiva della classificazione. Come misura di (im)purità degli attributi si è utilizzata l'entropia come richiesto nella consegna, ed il parametro considerato per la valutazione degli attributi è l'information gain (in realtà ho evitato il calcolo dell'information gain scegliendo direttamente l'attributo con remainder minore, osservando banalmente che se $(a - b) > (a - c)$, allora $b < c$ per $a, b, c > 0$, e in questo caso a, b, c sono tutti combinazioni lineari di valori di entropia, che nel nostro caso ha codominio $[0, 1]$).

Dovendo valutare sia valori discreti che valori numerici, ho deciso di utilizzare il pattern Strategy e creare una classe astratta `AttributeEvaluator`.
Tale classe `AttributeEvaluator` espone il metodo `get_evaluated_node()`, implementato in due sottoclassi:

- nella classe `DiscreteAttributeEvaluator`, il metodo calcola il resto in termini di entropia apportato dalla scelta di un attributo discreto, e successivamente restituisce il remainder associato;
- nella classe `NumericAttributeEvaluator`, il metodo prima genera tutti gli split point candidati (utilizzando la strategia suggerita in R/N §18.3.6), e successivamente restituisce, per ognuna delle divisioni valutate, quella che fornisce il minor remainder.

Una volta identificato l'attributo che garantisce la suddivisione più efficace, viene generato un oggetto `Node` per l'attributo scelto, che contiene una lista di oggetti `Branch`. Ogni branch ha un valore (la "label" del ramo), punta ad un altro nodo o ad una foglia, ed espone un metodo `compare(value)` che consente di testare un valore con quello della label. Il metodo è astratto ed è implementato nelle sottoclassi `EqualBranch` (per i branch che rappresentano l'operzione binaria di uguaglianza), `LessEqualBranch` (per l'operazione "$\leq$") e `GreaterBranch` (per l'operazione $>$).

La funzione PLURALITY-VALUE, che nel mio codice ho chiamato `get_most_frequent_value()`, banalmente opera scegliendo a maggioranza tra gli esempi rimasti. Nel caso in cui il metodo si trovi a dover generare una `Leaf` disponendo di esempi con lo stesso numero di casi positivi e negativi, sceglie casualmente il valore della foglia.

3 Pruning

La strategia utilizzata per il pruning è quella del *rule post-pruning*. Come evidente dal nome, si tratta di una tecnica di post-pruning: l'albero viene prima generato completamente, lasciando che si verifichi il fenomeno dell'overfitting, e successivamente vengono potati i suoi rami con minor capacità predittiva.

Con questo particolare metodo, l'albero viene convertito traducendo ogni cammino dalla radice alle foglie in una *regola*, ovvero una serie di condizioni in forma normale disgiuntiva (DNF). Ogni nodo viene convertito in una *precondition*, mentre la foglia viene convertita in una *postcondition*.

Dopo aver convertito l'albero in una lista di regole, si esamina ogni regola valutandone l'accuratezza predittiva su un validation set: per ogni regola, si calcola la precisione con cui la postcondition riesce a predire il valore corretto tra gli esempi del validation set che intraprendono quel cammino dalla radice alla foglia; dopodiché, si rimuovono ricorsivamente le preconditions la cui cancellazione riduce l'errore predittivo della regola.

Infine, si dispongono le regole in ordine decrescente rispetto al loro score di accuratezza. Per sfoltire ulteriormente l'albero finale, ho aggiunto un ulteriore step di pruning, in cui vengono rimosse anche le regole con capacità predittiva valutata nulla (potenzialmente generati da valori "rumorosi" del training set).

Da un punto di vista implementativo, le regole sono rappresentate come oggetti della classe `Rule`, e contengono una lista di `Preconditions` ed una `Postcondition`.
Così come la classe `Branch`, le `Preconditions` possono essere `EqualPrecondition`, `LessEqualPrecondition` e `GreaterPrecondition`, che rappresentano le diverse relazioni binarie.

Il metodo `prune_tree()` raggiunge ricorsivamente le foglie dell'albero, generando una nuova regola; ogni nodo tiene traccia delle regole generate dai propri nodi figli, e per ognuna di tali regole aggiunge in testa alle preconditions la condizione logica associata.

4 Risultati

4.1 Utilizzo del dataset

Le dimensioni dei set di validation e test cambiano in funzione della dimensione del training set: il validation set ha sempre 1/2 degli esempi rispetto al training set, mentre il test set ne ha 3/4.
Ad ogni iterazione dell'intero processo (generazione dell'albero dal training set—calcolo dell'accuratezza sul test set—pruning basato sull'errore sul validation set—nuovo calcolo dell'accuratezza sul test set) viene prelevato un campione casuale di esempi dai rispettivi dataset.

4.2 Impatto del pruning sulla capacità predittiva

L'algoritmo di pruning è la componente del modello che ha la complessità temporale maggiore. Infatti, non sono stati eseguiti test con training set maggiori di 5'000 elementi per il tempo eccessivo che richiedeva tale computazione. Altra osservazione da fare, seppur poco impattante sulla qualità complessiva del modello, è che la trasformazione dell'albero in una lista porta la complessità dell'operazione di ricerca all'interno della struttura dati ad $O(r \cdot a)$ nel caso peggiore, dove r è il numero di regole ed a il numero di attributi.

Tuttavia, pur pagando un alto costo computazionale durante il pruning, i benefici sull'accuracy di questa strategia si sono mostrati elevatissimi. L'applicazione al dataset Adult evidenzia come l'impatto della tecnica del rule post-

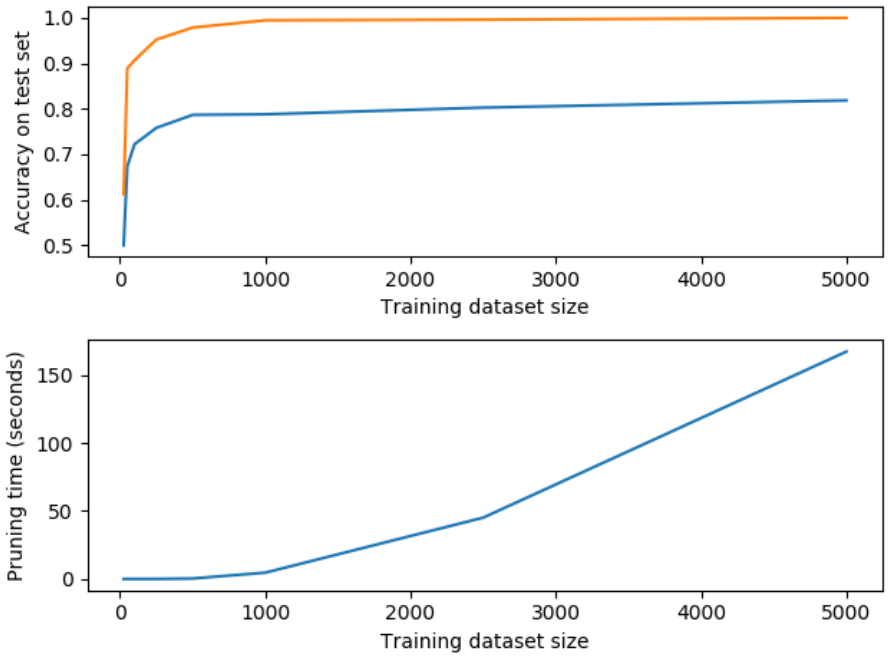

Figura 1: In blu l'accuracy prima del pruning, in giallo quella dopo il pruning

pruning sull'accuratezza predittiva sul test set sia sorprendentemente positivo anche utilizzando training set di dimensione relativamente piccola. Anche con training set di 50 elementi, il rule post-pruning consente di migliorare l'accuratezza anche di 20 punti percentuali. Con training set di dimensione più cospicua, l'accuratezza tende al 100%.

Training size	Acc. (before)	Acc. (after)	Pruning time
25	50%	61.23%	0.00
50	72.22%	88.89%	0.00
100	69.33%	94.67%	0.01
250	75.81%	95.23%	0.04
500	78.67%	99.26%	0.36
1000	78.80%	99.37%	4.63
2500	80.27%	99.43%	45.08
5000	80.85%	99.61%	167.11

Tabella 1: Accuratezza percentuale sul test set al crescere del training set

4.3 Conclusioni

La rule post-pruning si è dimostrata un'eccellente strategia per migliorare l'accuratezza della predizione in modo estremamente efficace, soprattutto nei casi in cui l'impatto del fenomeno dell'overfitting è maggiore. Tuttavia, il costo temporale dell'operazione di pruning suggerisce che è una strategia conveniente da

utilizzare soprattutto per combattere l'overfitting causato da scarsità di dati d'allenamento.