Report di statistica: XGBoost

BORELLI ROBERTO 147025 borelli.roberto@spes.uniud.it
SANTI ENRICO 148687 santi.enrico@spes.uniud.it

Corso di laurea Magistrale in Informatica
Università degli studi di Udine
Anno accademico 2022-2023

Indice

1 Sommario

L'obiettivo di questo progetto di laboratorio è quello di introdurre gli aspetti principali del metodo dei *gradient boosting tree*. Il contesto è quello dei metodi di assemblamento per gli alberi: nella sezione 2 mostreremo come combinare una serie di piccoli e semplici alberi di decisione per creare un predittore forte. Ci concentreremo poi (sezione 3) sulle caratteristiche principali della libreria R XGBoost che implementa i metodi sopra citati ed è considerata stato dell'arte sia dal punto di vista computazionale sia dal punto di vista applicativo statistico. Infine, nella sezione 4, mostreremo una semplice applicazione della libreria legata ad un caso di studio, traendo le dovute conclusioni.

2 Presentazione del metodo

2.1 Alberi di decisione

Introduzione. Gli alberi di decisione possono essere classificati come un modello di apprendimento supervisionato[1] in grado di partizionare in diverse categorie un dataset che, nel nostro contesto, può essere visto come un insieme di punti in uno spazio multi dimensionale. Questo processo di apprendimento può essere pensato come una procedura di *learning* di una formula logica CNF avendo a disposizione un insieme di dati etichettati. Da un punto di vista più formale abbiamo p predittori $X_1, \ldots, X_p$ e una variabile risposta Y (si noti che per ora non stiamo assumendo nulla sulle distribuzioni di tali variabili). Dato un dataset del tipo $(x_{11}, \ldots, x_{p1}, y_1), \ldots, (x_{1m}, \ldots, x_{pm}, y_m)$, un albero di decisione partiziona questi elementi (che chiameremo *sample*) separandoli in diversi nodi foglia. L'unione di tutte le foglie memorizza l'intero dataset e tutte le foglie sono mutuamente esclusive, ossia un sample è memorizzato in una e una sola foglia. Un nodo interno di un albero presenta due figli (destro e sinistro) e rappresenta una decisione o, in altri termini, un *if statment*, quindi una foglia può essere anche vista come la congiunzione delle decisioni prese nei nodi nel percorso radice-foglia. Gli alberi di decisione sono molto versatili ed in particolare possono essere usati sia nell'ambito della classificazione e sia in quello della regressione. Dato un nuovo sample (non etichettato) $x' = (x'_1, \ldots, x'_p)$ il nostro obiettivo è quello di assegnargli una label usando le informazioni presenti nell'albero costruito. Possiamo considerare una decisione presa al nodo i come un predicato $P_i(x)$ che restituisce vero o falso a seconda dei valori $x = (x_1, \ldots, x_p)$. Per predire la label di x' partiamo infatti dalla radice e valutiamo $P_{root}(x')$, se il valore restituito sarà vero, allora scenderemo nel figlio sinistro, altrimenti scenderemo nel figlio destro. In entrambi i casi iteriamo il processo a partire dal nodo in cui siamo scesi, valutiamo il nuovo predicato e scendiamo in un altro nodo. Quando siamo arrivati in un nodo foglia, siamo pronti ad assegnare la predizione $\tilde{y'}$ associata ad x'. Nel caso della regressione prenderemo come predizione la media delle label dei sample che troviamo nella foglia; nel caso della classificazione prenderemo la moda delle label. Un semplice albero di decisione è rappresentato in figura 1. Si noti che in figura sono presenti 6 foglie, quindi abbiamo partizionato lo spazio dei regressori in 6 regioni. Finora abbiamo solamente fornito un intuizione ad alto livello degli alberi di decisione, abbiamo visto come possano essere utilizzati ma non abbiamo detto nulla su come possano essere costruiti. Per trattare questi aspetti, formalizzeremo meglio i concetti nel prossimo paragrafo.

Contesto della stima di funzione. Il problema su cui ci focalizzeremo ora è quello della stima di funzione [Natekin and Knoll, 2013]. Dato un insieme di m sample $(x_1, \ldots, x_m)$ dove ogni x_i è del tipo $x_i = (x_{1i}, \ldots, x_{pi})$ e un insieme di m etichette associate ai sample $(y_1, \ldots, y_m)$ il nostro obiettivo è quello di ricostruire la ignota dipendenza funzionale $x \xrightarrow{f} y$. Stimeremo quindi una funzione $\hat{f}(x)$ minimizzando una specifica funzione di costo $\psi(y, f(x))$. Sceglieremo dunque la nostra stima ponendo $\hat{f}(x) = \arg\min_{f(x)} \psi(y, f(x))$. La variabile risposta Y può appartenere a varie distribuzioni e in base alla distribuzione dovrà essere utilizzata un'adeguata funzione di costo. Ad esempio nel caso Y sia una variabile continua potremmo utilizzare il metodo dei minimi quadrati.

[1] Un modello in cui i dati di training necessitano essere etichettati

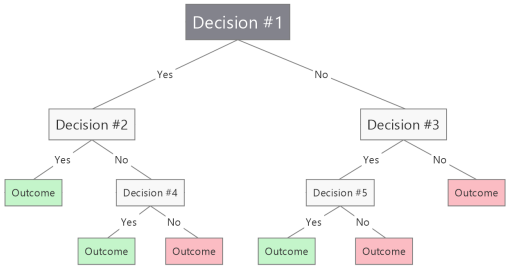


Figura 1: La struttura di un albero di decisione

Per trattare il problema possiamo restringerci considerando $\hat{f}(x)$ come appartenente alla famiglia di funzioni parametriche del tipo $f(x, \theta)$. In questo modo siamo passati da un problema di ottimizzazione, in particolare di minimizzazione, ad un problema di ricerca del miglior parametro θ.

Per stimare θ utilizziamo una procedura iterativa. Date j iterazioni la stima $\hat{\theta}$ sarà la somma delle m stime $\hat{\theta}_i$ effettuate ad ogni passo i.

Il metodo più usato per questo task è lo *steepest gradient descent*, ovvero una procedura di ottimizzazione basata sul raffinamento iterativo della funzione di costo seguendo la direzione del gradiente della stessa. Il problema del metodo del gradiente sta nel fatto che potrebbe trovare un ottimo locale e non globale.

2.2 Metodi di assemblamento, boosting e gradient boosting

Assemblamento. L'idea alla base di un metodo di assemblamento è quella di utilizzare diversi modelli di regressione o classificazione semplici per costruire un predittore forte. E' dunque sufficiente considerare una serie di predittori deboli e combinarli.

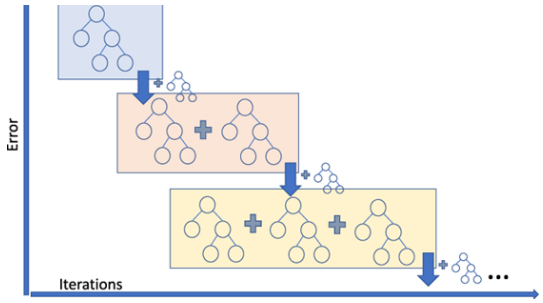


Figura 2: Assemblamento di alberi

In figura 2 è possibile notare un assemblamento di alberi: da un predittore debole che presenta un errore elevato, creiamo una serie di alberi che via via riducono l'errore sempre più.

Boosting. Il boosting è un caso particolare dei metodi di assemblamento ed è un processo iterativo in cui all'iterazione *i-ma* stimiamo la funzione f_i come la somma della funzione ottenuta all'iterazione precedente e il valore di un predittore debole ossia $\hat{f}_t \leftarrow \hat{f_{t-1}} + h(x, \theta_t)$. In particolare vogliamo che il nuovo predittore ottenuto abbia correlazione massima con il valore della funzione di costo.

Come predittori deboli possiamo utilizzare gli *stump*. Gli stump sono alberi di decisione che contengono la radice, e due sole foglie, senza nessun altro nodo interno. Si nota immediatamente che gli stump possono estrapolare solo funzioni con un valore costante e non possono approssimare correttamente nemmeno una funzione lineare.

In figura 3 possiamo vedere l'idea alla base della procedura di boosting. Da un modello debole, calcoliamo gli errori, costruiamo un modello (debole) che migliori gli errori e lo aggiungiamo al modello principale rafforzandolo.

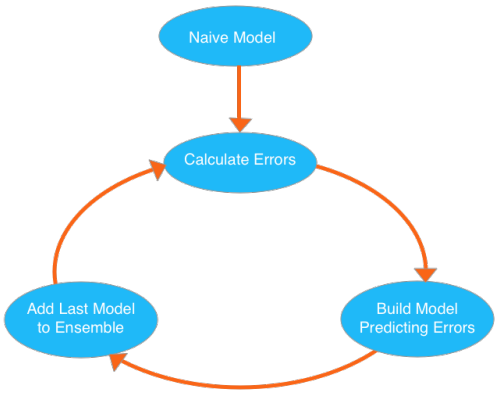


Figura 3: L'idea alla base della procedura di boosting

Gradient tree boosting. Nel gradient tree boosting ad ogni passo costruiamo un nuovo predittore considerando il predittore precedente unito ad uno stump allenato sui residui precedenti, tale stump è ottenuto usando il gradiente della funzione di costo. Abbiamo

$$Stump_i \leftarrow -\frac{\partial Cost(\hat{f_{i-1}}, y)}{\partial \hat{f_{i-1}}} \tag{1}$$

$$\hat{f}_t \leftarrow \hat{f_{t-1}} + Stump_{i-1} \tag{2}$$

La terminazione dell'algoritmo di boosting è garantita dal ciclo enumerativo per calcolare i diversi $Stump_i$. All'aumentare delle iterazioni la precisione (nella predizione sui sample usati per l'allenamento) e la complessità del modello aumentano, introducendo però un problema di overfit sempre maggiore, che farà risentire la precisione delle predizioni. Per questo motivo risulta importante scegliere un criterio di stop adeguato per la procedura descritta. Un altro approccio che aiuta ad evitare l'overfitting è il così detto *shrinkage*, ovvero l'aggiunta di un peso η all'equazione 2:

$$\hat{f}_t \leftarrow \hat{f_{t-1}} + \eta Stump_{i-1} \tag{3}$$

Lo shrinkage riduce l'influenza di ogni albero individuale lasciando margine di miglioramento per gli alberi successivi. Una scelta tipica è porre $\eta = 0.5$.

Generalizzando quanto visto sopra, al posto degli stump potremmo utilizzare dei generici alberi di decisione come predittori deboli. La libreria XGBoost che analizzeremo nelle sezioni successive, mette a disposizione un parametro `max_depth` che ci permette di definire l'altezza massima di ogni albero utilizzato come predittore debole nelle varie iterazioni.

3 Presentazione della libreria

Come riportato dalla documentazione [2], XGBoost è una libreria con una forte attenzione all'ottimizzazione, dal momento che è stata progettata per essere utilizzata su dataset complessi, che fornisce una serie di metodi legati al gradient boosting per l'implementazione di modelli di regressione e classificazione. E' supportata da diversi linguaggi e permette di manipolare dataset in diversi formati, come ad esempio i DataFrames della libreria Pandas (in python). Un primo esempio introduttivo all'utilizzo della libreria in R è reperibile alla pagina [3].
Per comprendere il funzionamento di molte delle funzioni della libreria è necessario conoscere il meccanismo delle callback, presentato nel prossimo paragrafo.

[2] `https://xgboost.readthedocs.io/en/stable/`
[3] `https://xgboost.readthedocs.io/en/stable/R-package/xgboostPresentation.html`

3.1 Parallelismo

L'attenzione al parallelismo e all'ottimizzazione sono feature centrali di questa libreria. Per quanto riguarda il parallelismo, è immediato notare che molteplici funzioni ammettono come argomento opzionale un valore che specifica il numero di thread con i quali eseguire il compito specificato (e.g. il parametro `nThread` nella funzione `xgb.train`). Nel paragrafo 4.6 è presentato come la scelta del numero di thread possa influenzare sensibilmente i tempi di esecuzione di script basati su questa libreria.

3.2 Il sistema di callback

La libreria XGBoost mette a disposizione un sistema di callback che possono essere richiamate durante o alla fine di un'iterazione di boosting. Come vedremo in seguito nella lista delle funzioni principali le callback presentano un nome che inizia con `cb.` e offrono dei servizi di uso comune, ad esempio potremmo essere interessati a salvare il modello ad ogni iterazione del boosting e questo può essere fatto con una specifica callback. Le callback possono essere anche *user defined* consentendo del codice più modulare. Come implementazione standard le callback sono eseguite dopo ogni iterazione di boosting; per fare sì che vengano eseguite prima dell'iterazione è possibile utilizzare il parametro di `is_pre_iteration`. Usando invece il parametro `finalize`, la parte finale della callback verrà eseguita solo quando tutto il processo di boosting è stato completato.

3.3 Funzioni principali

Sono in seguito elencate e descritte alcune delle funzioni principali (utilizzate nel caso di studio presentato successivamente) del pacchetto XGBoost [Chen et al., 2019]. Le funzioni sono state raggruppate logicamente ed oltre ad una descrizione dei vari parametri e del risultato atteso, per i metodi principali viene anche fornito un tipico caso d'uso. Per le funzioni più complesse vengono inoltre forniti dei piccoli listati di codice. Questa sezione non vuole essere un manuale esaustivo delle funzioni, al contrario ci proponiamo di presentarne solamente alcune procedure che risultano essenziali per l'utilizzo il pacchetto XGBoost e che dovranno essere note nell'analisi del caso di studio che affronteremo nella sezione 4.

Rappresentazione interna. Iniziamo descrivendo alcune funzioni per la manipolazione e gestione dei tipi di dato che la libreria definisce.

- **xgb.save**: Si tratta di una funzione base per salvare oggetti manipolati dalla libreria ossia oggetti di tipo `xgb.Booster`. Si noti che R mette a disposizione la funzione `saveRDS` per salvare dei generici oggetti R. La differenza principale rispetto alle funzione built-in è che `xgb.save` garantisce la compatibilità tra varie versioni di XGBoost ed è quindi da preferirsi nel caso in cui vi sia un modello M allenato con una versione x di XGBoost e si sia interessati a manipolare M in una versione di XGBoost successiva ad x.

- **xgb.DMatrix**: Funzione necessaria per convertire una matrice in una `DMatrix`, un formato standard utilizzato dalle funzioni della libreria per manipolare i dati. Può essere utilizzata anche per convertire direttamente una matrice contenuta in un file.

- **xgb.load**: E' la funzione complementare di `xgb.save`, ci permette di caricare un modello salvato su un file. Si noti che non può essere usata per caricare un generico oggetto R.

- **dim.xgb.DMatrix**: Funzione che fornita in input una `DMatrix` ne restituisce le dimensioni (e.g. il numero di righe e di colonne).

- **xgb.DMatrix.save**: Funzione che data una `DMatrix` ed una stringa, salva in formato binario la matrice nel file specificato dalla stringa. E' particolarmente utile nel caso durante l'analisi si abbia a che fare con matrici di grandi dimensioni e si voglia convertirle una sola volta in `DMatrix`. Il formato dei dati salvati è diverso da una `DMatrix`.

- **dimnames.xgb.DMatrix**: Funzione che permette di gestire i nomi delle colonne per oggetti di tipo `xgb.DMatrix` analogamente alle funzioni di R `rownames` e `colnames` applicate alle matrici.

Callbacks. Vengono di seguito descritte alcune delle callbacks principali.

- **cb.gblinear.history**: Funzione di callback che restituisce ad ogni iterazione il valore dei parametri del modello. Funzione particolarmente utile, come la successiva, per analizzare il miglioramento del modello durante il training.
- **cb.print.evaluation**: Callback che dato un valore intero i permette di stampare il risultato della valutazione del modello ogni i iterazioni. Callback particolarmente utile per comprendere come il modello stia migliorando durante la fase di training.
- **cb.reset.parameters**: Funzione di callback che riceve in input una lista di elementi, contenente i parametri da modificare all'inizio di ogni iterazione. Tali elementi comprendono il nome del parametro ed il valore che essi dovranno assumere all'inizio di ogni iterazione. Un esempio può essere legato alla modifica, durante diverse iterazioni dello *shrinkage* η.
- **cb.save.model**: Callback che permette di salvare su file un modello creato o in fase di creazione. Risulta essere particolarmente utile nel caso di modelli complessi che richiedono molteplici iterazioni. Puo' essere utilizzata per eseguire una singola volta il training del modello. E' infatti possibile specificare il numero di iterazioni ogni qual volta si vuole salvare il modello ottenuto (il salvataggio è quindi periodico). Nel caso di salvataggio periodico è possibile sovrascrivere il salvataggio precedente oppure è possibile creare una sequenza di salvataggi progressivi nominati ad esempio `model_i.model` dove i rappresenta il numero di iterazioni dopo il quale si è salvato il modello. Il caso in cui vengano salvati una serie di modelli periodicamente risulta interessante per lo studio dell'overfit, in quanto è possibile allenare il modello su un numero elevato di iterazioni, che verosimilmente risulterà essere in overfit rispetto al training set. In seguito sarà quindi possibile visualizzare i modelli ad iterazioni precedenti e trovare un boundary adeguato al numero di iterazioni.

Learning, training and prediction. Le funzioni descritte in questo paragrafo costituiscono il cuore della libreria e sono le funzioni che implementano il training dei modelli e diversi metodo statistici.

- **xgb.train**: Due delle fasi di maggiore importanza nell'ambito del machine learning sono quelle legate alla definizione e allenamento del modello. La funzione train del pacchetto XGBoost è ultizzata allo scopo di fornire un'interfaccia con un livello elevato di astrazione all'utente per definire (dunque allenare) un albero di decisione. Trattandosi di una delle funzioni centrali e di maggior rilievo nella libreria essa prevede in input molteplici parametri, di cui molti opzionali. Tra i principali vi sono:
  - `data`: I dati di training (sottoforma di `DMatrix`). Il dataset va dunque diviso precedentemente nella parte di training e test.
  - `nrounds`: Specifica il numero delle iterazione di boosting da eseguire.
  - `feval`: l'handle di una funzione con due parametri (il valore della risposta predetta dal modello, ed il dataset di training). La funzione è utilizzata per la valutazione del modello ad ogni iterazione.
  - `watchlist`: Una lista (genereralmente breve) di dataset (`DMatrix` con un ridotto numero di righe rispetto al dataset di training), utlizzati per la valutazione dei modelli generati nelle diversi fasi di boosting. Sono dunque da intendersi come una lista di validation set.
  - `obj`: L'handle della funzione obiettivo che il modello deve ottimizzare.

  Vi sono poi una serie di parametri (opzionali) legati al salvataggio dei vari modelli (al termine di un certo valore di iterazioni), ed al miglioramento di un modello precedentemente creato:

- `save_period`: Un valore numerico che specifica il numero di interazioni dopo le quali salvare nuovamente il modello.
- `save_name`: Una stringa contenente il prefisso del nome con cui salvare il modello creato.
- `xgb_model`: Un modello precedentemente creato del quale è possibile continuare l'allenamento.

infine vi è un parametro `params` che consente la specifica di una lista di coppie `nome_parametro = valore` per definire una serie di parametri aggiuntivi, reperibili dalla documentazione [4].

- **predict.xgb.Booster**: E' la funzione che dati in input il modello (o un suo handle) ed una matrice `xgb.DMatrix`, calcola le predizioni sui dati nella matrice in base al modello specificato. Un parametro opzionale interessante è rappresentato da `iterationrange` che specifica tramite un vettore di 2 interi[5] quali livelli dell'albero utilizzare per la predizione dell'output. Anche in questo caso la funzione può essere utilizzata in diversi scenari, si pensi ad esempio al caso in cui si è allenato il modello con un numero molto alto di iterazioni x ed avendo in seguito scoperto che il valore ottimo sarebbe stato $y << x$ (siamo dunque overfit), possiamo immediatamente utilizzare il modello per le predizioni con soli y livelli e senza bisogno di allenare nuovamente il modello che può essere un'operazione molto costosa.

  Per chiarire l'utilizzo vediamo e commentiamo un semplice esempio di codice.

```r
data(agaricus.train, package='xgboost')
train <- agaricus.train
bst <- xgboost(data = train$data, label = train$label, max_depth = 2,
  eta = 0.5, nthread = 2, nrounds = 5, objective = "binary:logistic")
predict(bst, train$data)
predict(bst, train$data, iterationrange = c(1, 2))
```

  Per prima cosa viene caricato un dataset di esempio della libreria XGBoost. Viene in seguito creato un modello di decisione dove il learning rate vale 0.5 ed ogni albero base avrà un altezza massima di 2. Il modello di decisione risultante sarà composto da 5 alberi. Nella penultima riga viene richiamata la funzione predict sul training set senza particolari parametri. Nell'ultima riga invece specifichiamo che vengano utilizzati solamente i primi due alberi. Confrontando le predizioni sugli stessi esempi si nota che queste sono diverse. Si noti infine che avendo creato l'albero con 5 livelli, se avessimo posto `iterationrange=c(1,6)` la predizione avrebbe utilizzato i tree nell'intervallo $[1, 6)$ ossia il modello completo.

- **xgb.attr**: Ci permette di salvare dei metadati relativi ad un modello XGBoost. Si noti che R presenta già un concetto di attributi relativi ad un oggetto e in particolare possiamo aggiungere attributi ad un modello XGBoost. La differenza è che i metadati memorizzati con `xgb.attr` saranno conservati al momento del salvataggio del modello con `xgb.save` mentre non saranno conservati gli attributi memorizzati secondo le modalità standard di R.

- **xgb.config**: Ci permette di accedere ai parametri e alle informazioni del modello come una stringa in formato JSON. Funzione particolarmente importante per quanto riguarda l'interoperabilità dei dati e quindi per tutte le tipologie di applicazioni che necessitano un formato comune di scambio dati.

- **xgb.create.features**: Funzione che ci permette di migliorare il learning aggiungendo delle particolari nuove features al training set sulla base di un gradient boosting tree precedentemente calcolato. In particolare, avendo a disposizione un dataset D, possiamo costruire il modello M con la classica funzione `xgb.train`. Dati in input M e D la funzione restituisce un nuovo dataset D' (sotto forma di matrice) composto dalle features originali più alcune features aggiunte. Vediamo ogni albero in M (il modello precedentemente calcolato) come una variabile categoriale

[4] https://xgboost.readthedocs.io/en/latest/parameter.html
[5] L'intervallo sarà del tipo $[a, b)$

che ha come valore su un sample, l'indice della foglia su cui cade tale sample. Questa variabile categoriale viene codificata con varie features binarie. Il dataset D' sarà quindi composto da tutte le fetaures del dataset originale unite a tutte le features binarie ottenute considerando ogni albero del modello M come fattori a k_i livelli (dove k_i è il numero di foglie dell'albero i-mo in M).

Ottenuto D' è naturale calcolare un nuovo modello M' che in molte situazioni avrà prestazioni migliori, come spiegato in [He et al., 2014].

Anche in questo caso presentiamo un semplice esempio per chiarire meglio il concetto.

```
data(agaricus.train, package='xgboost')
param = list(max_depth=2, eta=1, objective='binary:logistic')

D = with(agaricus.train, xgb.DMatrix(data, label = label))
M = xgb.train(params = param, data = D, nrounds = 5, nthread = 2)

new_dataset = xgb.create.features(model = M, agaricus.train$data)
D_prime = xgb.DMatrix(data = new_dataset, label = agaricus.train$label)

M_prime = xgb.train(params = param, data = D_prime, nrounds = 5, nthread = 2)
```

Nell'esempio, carichiamo il dataset nella variabile D, calcoliamo un modello M con massima altezza 2. Utilizziamo quindi la funzione `xgb.create.features` per creare un nuovo dataset D_{prime}. Si noti che con i parametri così settati M è composto da 5 alberi ognuno dei quali ha al massimo 4 foglie, quindi il dataset D_{prime} conterrà al massimo 20 feaures in più rispetto a D. Calcoliamo infine, nella maniera usuale il modello M_{prime}.

- **xgb.cv**: E' la funzione di cross validation della libreria XGBoost. La cross validation è una metodologia generica che ci permette di valutare delle metriche relative al modello che non siano troppo ottimistiche e che dipendano il meno possibile dal training set utilizzato. Nella cross validation il dataset viene diviso in k insiemi (detti *fold*) e ognuno viene utilizzato a turno come test set. Quando $k = 1$ la procedura prende il nome di *leave-one-out-cross-validation*. E' importante notare che la funzione non prende in input un modello già creato ma prende in input parametri molto simili a quelli di `xgb.train` in aggiunta al parametro che specifica il numero di folds da utilizzare `nfold`. Notiamo infine che la funzione `xgb.cv` non restituisce un modello (e quindi non potremo immediatamente applicare la funzione `xgb.predict` sul risultato del modello). Infatti tale funzione non serve per allenare un modello ma serve solamente per calcolare delle metriche. Per una descrizione completa di tutti i parametri (che in questo caso sono molti) suggeriamo la guida ufficiale. Anche per questo metodo particolarmente importante forniamo un esempio di utilizzo.

```
data(agaricus.train, package='xgboost')
dtrain<-with(agaricus.train,xgb.DMatrix(data, label=label))
cv<-xgb.cv(data = dtrain, nrounds = 4, nfold = 5,
        metrics = list("mae","auc"), max_depth = 3,
        eta = 1, objective = "binary:logistic")
print(cv$evaluation_log)
```

Viene eseguita la procedura di cross validation sulla `xgb.DMatrix` `dtrain`. Costruiamo dunque un predittore binario che utilizzerà 4 composizioni e l'albero costruito ad ogni iterazione avrà altezza al massimo 3. Il numero di fold è 5 ossia `dtrain` sarà spezzato in 5 parti di cui a turno, 4 saranno usate per allenare il modello descritto, e la rimanente sarà utilizzata per valutare le metriche specificate, in questo caso `mae` e `auc`. Stampando `cv$evaluation_log` verrà mostrata una tabella con le metriche specificate valutate nella procedura di cross validation e saranno messe in relazione con le iterazioni di boosting (in questo caso 4).

- **xgb.importance**: Funzione che fornito in input un modello restituisce, qualora esso sia un albero, una tabella contenente tra i valori più rilevanti i nomi delle variabili (feature) utilizzate nel modello. Un altro dato contenuto nell'output è per ogni feature il proprio contributo al modello. La funzione accetta in input anche modelli lineari.

Funzioni di output Descriviamo ora funzioni di output sia testuale che grafico.

- **print.xgb.Booster**: Funzione che consente di stampare informazioni di un oggetto `xgb.Booster`. In particolare, dato in input un modello su cui abbiamo eseguito il training, la funzione stampa i diversi parametri di training tra cui anche la lista delle callbacks associate. Stampa inoltre per ogni iterazione la *log loss* riferita al training set, questa informazione può essere utile per capire quando il modello inizia ad entrare in overfitting.
- **getinfo**: Funzione basilare per ottenere le informazioni da oggetto di classe `xgb.DMatrix`. Può essere paragonata alla funzione R `str` applicata ai dataframe.
- **xgb.ggplot.deepness**: Stampa due grafici relativi alla profondità delle foglie in un modello. In particolare sono stampate le seguenti due distribuzioni rispetto alla profondità delle foglie:
  - La distribuzione del numero di foglie ad una certa profondità.
  - La distribuzione del numero medio di osservazioni che cadono in foglie ad una profondità fissata.

  Questi grafici possono essere utili per determinare il valore ottimale del parametro `max_depth` usato in fase di creazione del modello. In figura 4 si vede un esempio di output della funzione.

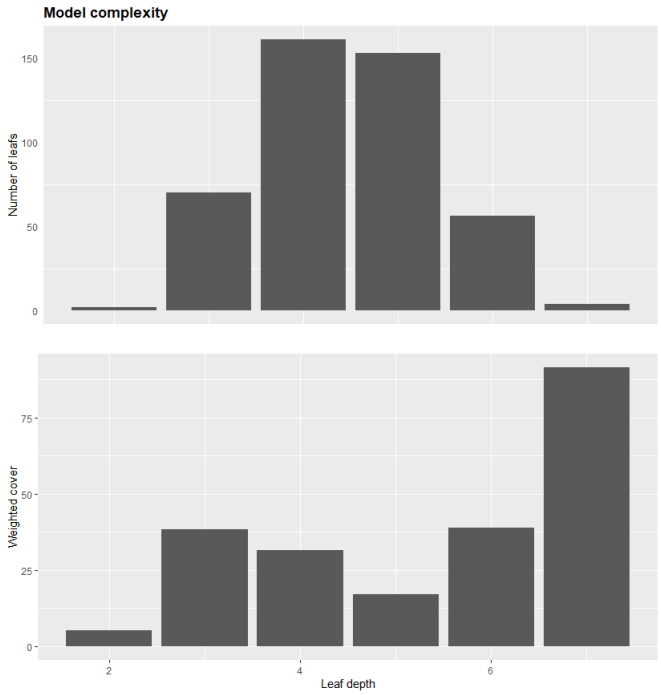


Figura 4: Grafico stampato dalla funzione `xgb.ggplot.deepness`

- **xgb.plot.tree**: Stampa il grafico relativo alla struttura di un gradient boosting tree. Ogni nodo contiene le seguenti informazioni:
  - Feature name: il nome della feature sulla quale il nodo agisce.

– Cover. Intuitivamente corrisponde al numero di esempi di training visti da quel nodo. Se si tratta di una foglia è il numero di esempi di training collezionati nella foglia
– Gain (per nodi interni): Corrisponde all'importanza del nodo nel modello.
– Value (per nodi foglia): Il valore marginale con cui la foglia contribuisce alla predizione finale.

In figura 5 vediamo un esempio di un modello composto da 2 alberi con parametro `max_depth` pari a 2.

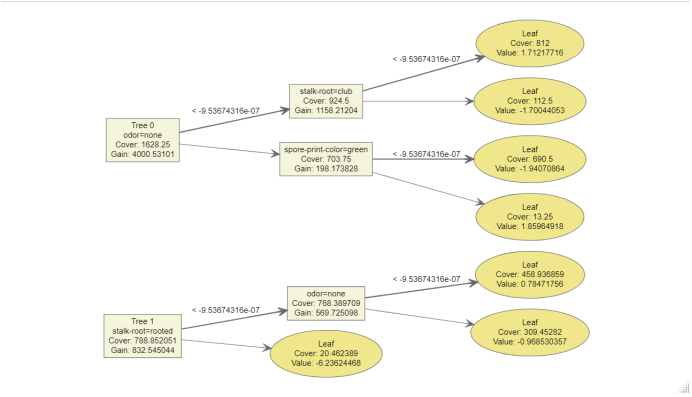


Figura 5: Grafico stampato dalla funzione `xgb.plot.tree`

4 Caso di studio

4.1 Presentazione del dataset: Glass Identification

Introduzione. Come caso di studio, abbiamo scelto il *Glass Identification Data Set* reperibile dalla repository UCI[6]. Si tratta di un dataset abbastanza consolidato (risale al 1987) introdotto nel paper [Evett and Spiehler, 1989] in ambito criminologico. In particolare spostandoci sulla scena del crimine, quando si trovano dei frammenti di vetro (anche piccoli) vengono misurate alcune caratteristiche fisiche, e come obbiettivo vi è quello di capire quale sia l'origine dello stesso (ad esempio capire se provenga da una bottiglia rotta, una finestra, ecc...). La classificazione del frammento di vetro potrà quindi venire utilizzata dalle forze dell'ordine come indizio utile per le successive indagini. Le proprietà facilmente misurabili del vetro sono le seguenti:

- Indice di rifrazione (RI)
- Composizione del materiale

Concentrandoci ora sulle tipologie di oggetti di vetro, già ad un primo sguardo si percepisce che sono troppe e si necessita di uno schema semplificato di classificazione. La distinzione più rilevante in questo caso è quella tra *window-glass* e *non-window-glass*, che specifica se il frammento provenga da una finestra o meno. La prima categoria si può ulteriormente dividere in 2 sottocategorie *building-window* e *vehicle-window* in base alla provenienza della finestra e ciascuna delle 2 sottocategorie può essere a sua volta divisa in altre due, ovvero *float* e *non-float* in base al processo con cui il vetro è stato realizzato. Per la seconda categoria, possiamo distinguere vetri provenienti da *Container*, *Tableware* e *Headlamp*. Per quanto riguarda la composizione del materiale, diversi elementi sono considerati, e sono presentati nella composizione del dataset.

Composizione del dataset. Il Glass Identification dataset (contenuto nel file `glass.data`) contiene 214 sample, ognuno con 10 caratteristiche:

[6] `https://archive.ics.uci.edu/ml/datasets/Glass+Identification`

1. RI: indice di rifrazione
2. Na: Sodio
3. Mg: Magnesio
4. Al: Alluminio
5. Si: Silicio
6. K: Potassio
7. Ca: Calcio
8. Ba: Bario
9. Fe: Ferro
10. Type: Tipo di vetro

Le variabili 2-9 riguardano la composizione del frammento di vetro[7] L'attributo 10 è il tipo di vetro, codificato con un numero:

1. building-window-float
2. building-window-non-float
3. vehicle-window-float
4. vehicle-window-non-float
5. Container
6. Tableware
7. Headlamp

Nel dataset non appare nessun sample con tipologia di vetro pari a 4.

Lavori correlati. Originariamente il dataset è stato analizzato in [Evett and Spiehler, 1989] con il pacchetto BEAGLE e confrontato con i metodi statistici *Nearest-Neighbours* e *Discriminant Analysis*. In base ai peper riportati sul sito UCI, il dataset è stato analizzato con il metodo *Boosting-nearest-neighbor* [Athitsos and Sclaroff, 2005] e con alberi di decisioni classici [Bioch et al., 1997]. Sempre in base a quanto riportato sul sito UCI e al meglio delle nostre ricerche il dataset non risulta essere stato analizzato utilizzando la metodologia dei gradient boosting tree presentata in sezione 2, dunque analizzare il dataset e sperimentare su esso con la libreria XGBoost ci è sembrato opportuno.

4.2 Analisi dati esplorativa

Iniziamo con il caricare il dataset e rimuoviamo la prima colonna in cui sono memorizzati gli indici dei sample.

```
glass = read.table("glass.data", sep=",", header=F)[-1]
colnames(glass) = c("RI","Na","Mg","Al","Si","K","Ca","Ba","Fe","Type")
str(glass)

## 'data.frame': 214 obs. of  10 variables:
##  $ RI  : num  1.52 1.52 1.52 1.52 1.52 ...
##  $ Na  : num  13.6 13.9 13.5 13.2 13.3 ...
##  $ Mg  : num  4.49 3.6 3.55 3.69 3.62 3.61 3.6 3.61 3.58 3.6 ...
##  $ Al  : num  1.1 1.36 1.54 1.29 1.24 1.62 1.14 1.05 1.37 1.36 ...
##  $ Si  : num  71.8 72.7 73 72.6 73.1 ...
##  $ K   : num  0.06 0.48 0.39 0.57 0.55 0.64 0.58 0.57 0.56 0.57 ...
##  $ Ca  : num  8.75 7.83 7.78 8.22 8.07 8.07 8.17 8.24 8.3 8.4 ...
##  $ Ba  : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ Fe  : num  0 0 0 0 0 0.26 0 0 0 0.11 ...
##  $ Type: int  1 1 1 1 1 1 1 1 1 1 ...
```

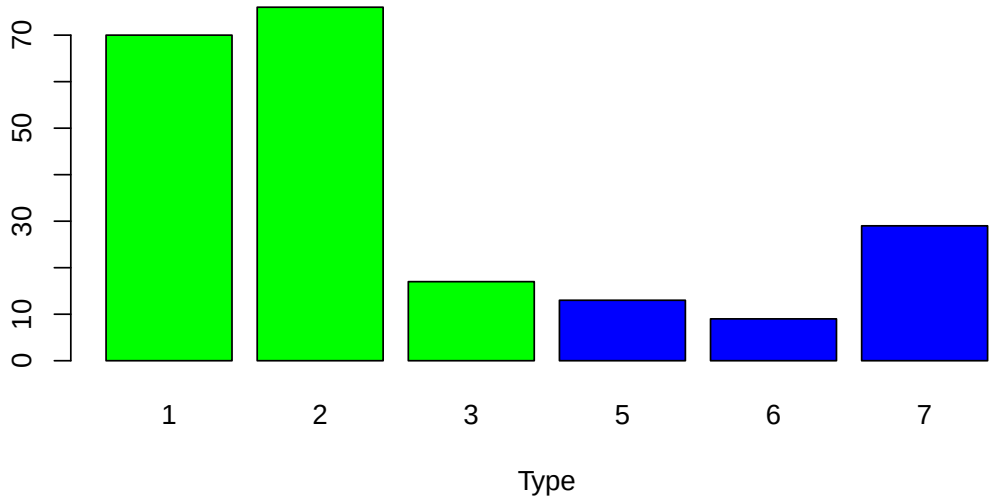
La colonna 'Type' è di tipo `int` e con il seguente codice la trasformiamo (casting) in variabile categoriale. Per comodità, creiamo la variabile (colonna) 'Window' che ha avrà valore `TRUE` su tutti i sample il cui vetro proviene da una finestra.

[7] Unità di misura: peso percentuale nel corrispondente ossido.

```
glass$Window = (glass$Type > 4)
glass$Window = factor(glass$Window)
levels(glass$Window) = c(TRUE,FALSE)
glass$Type = factor(glass$Type)
levels(glass$Type) = c(1,2,3,5,6,7)
```

Iniziamo l'analisi con il presentare con un barplot la distribuzione dei sample rispetto al tipo di vetro. Le tipologie da 1 a 4 rappresentano vetro di tipo `window` e li rappresentiamo in verde, in blue invece rappresentiamo i frammenti di vetro `non-window`. Si vede subito che siamo di fronte ad un dataset sbilanciato a favore dei tipi `window`.

```
t = table(glass$Type)
barplot(t, xlab="Type", col=c("green","green","green","blue","blue","blue"))
```



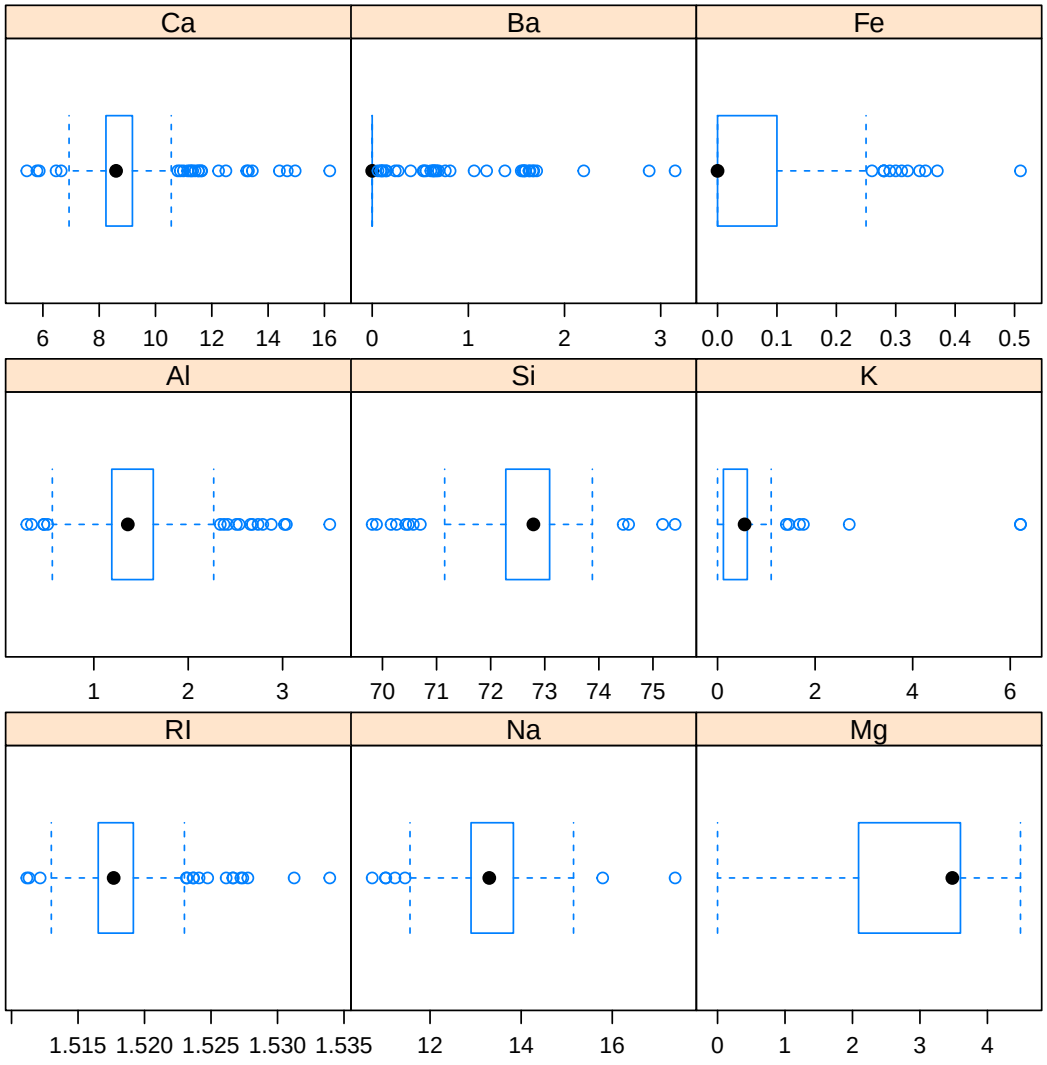
Nel dataset troviamo 163 esempi di tipo `window` e 51 esempi di tipo `non-window`.

```
table(glass$Window)

##
##  TRUE FALSE
##   163    51
```

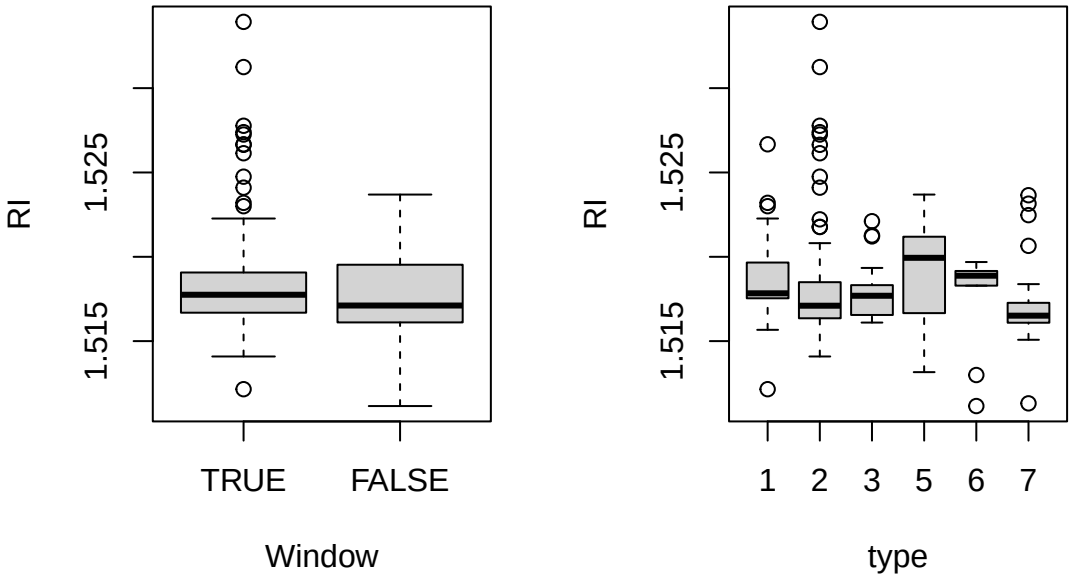
Com'è possibile notare i frammenti (in base alla categoria) non presentano una distribuzione uniforme. Mostriamo ora i vari boxplot delle variabili esplicative candidate.

```
library(lattice)
bwplot(~ RI + Na + Mg + Al + Si + K + Ca + Ba + Fe,
       data=glass, ylab="", outer = TRUE, scales = list(x = "free"),
       xlab="", layout=c(3,3), main="", aspect="fill")
```



Notiamo che i dati sono su scale molto differenti, quindi saranno necessarie opportune trasformazioni. Inoltre diversi possibili outlier sono segnalati. Sono di seguito riportati inoltre alcuni conditional boxplot realizzati per meglio comprendere come i diversi valori dell'indice di rifrazione siano propri di diverse tipologie di vetro, così come la presenza dei diversi elementi.

```
par(mfrow = c(1,2))
boxplot(glass$RI~glass$Window,xlab="Window",ylab="RI")
boxplot(glass$RI~glass$Type,xlab="type",ylab="RI")
```

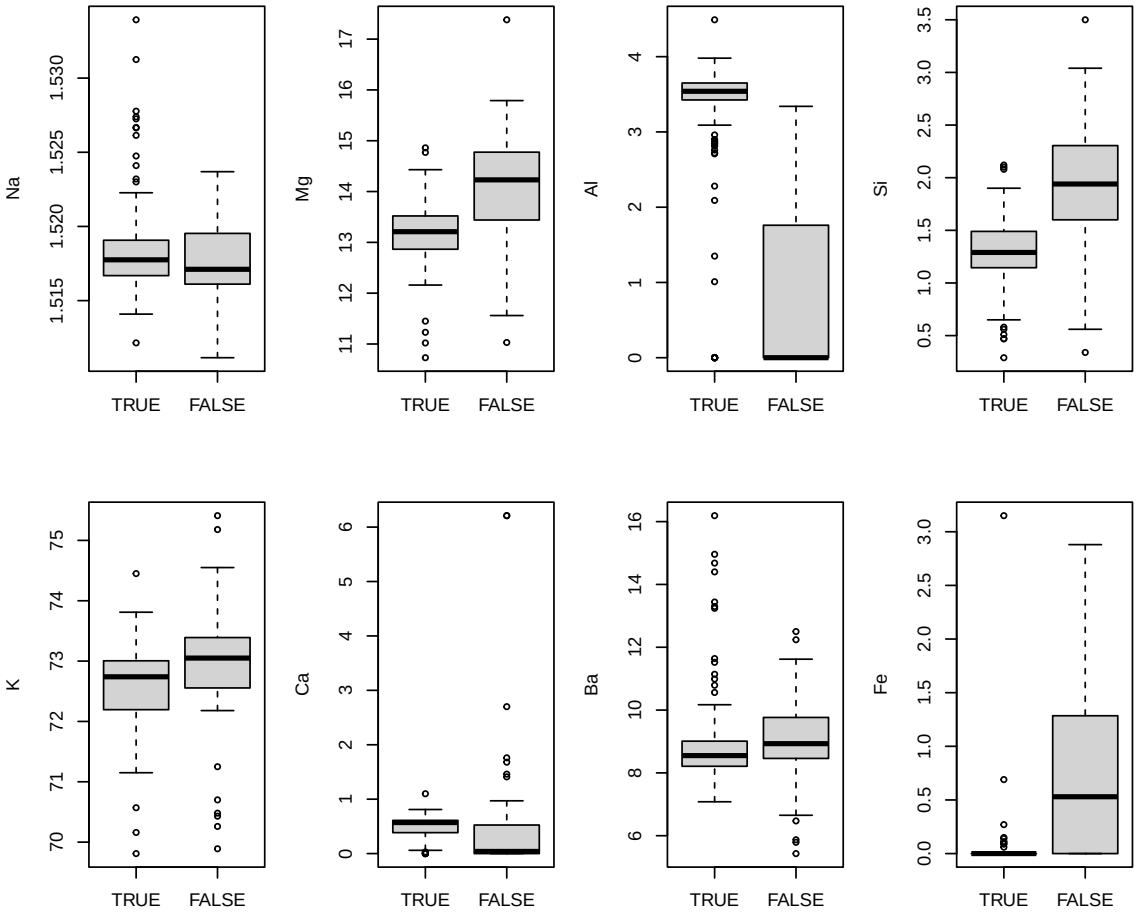



```
par(mfrow = c(1,1))
```

Dal primo boxplot sembrerebbe che il valore dell'indice di rifrazione a seconda del fatto che il vetro provenga da una finestra o meno presenti una minor variabilità e una mediana leggermente più alta nel primo caso. Segue un boxplot più dettagliato (i.e. sui singoli livelli di 'Type'), dal quale possiamo evincere che la variabilità riferita ad 'RI' aumenta o diminuisce a seconda della provenienza del vetro.

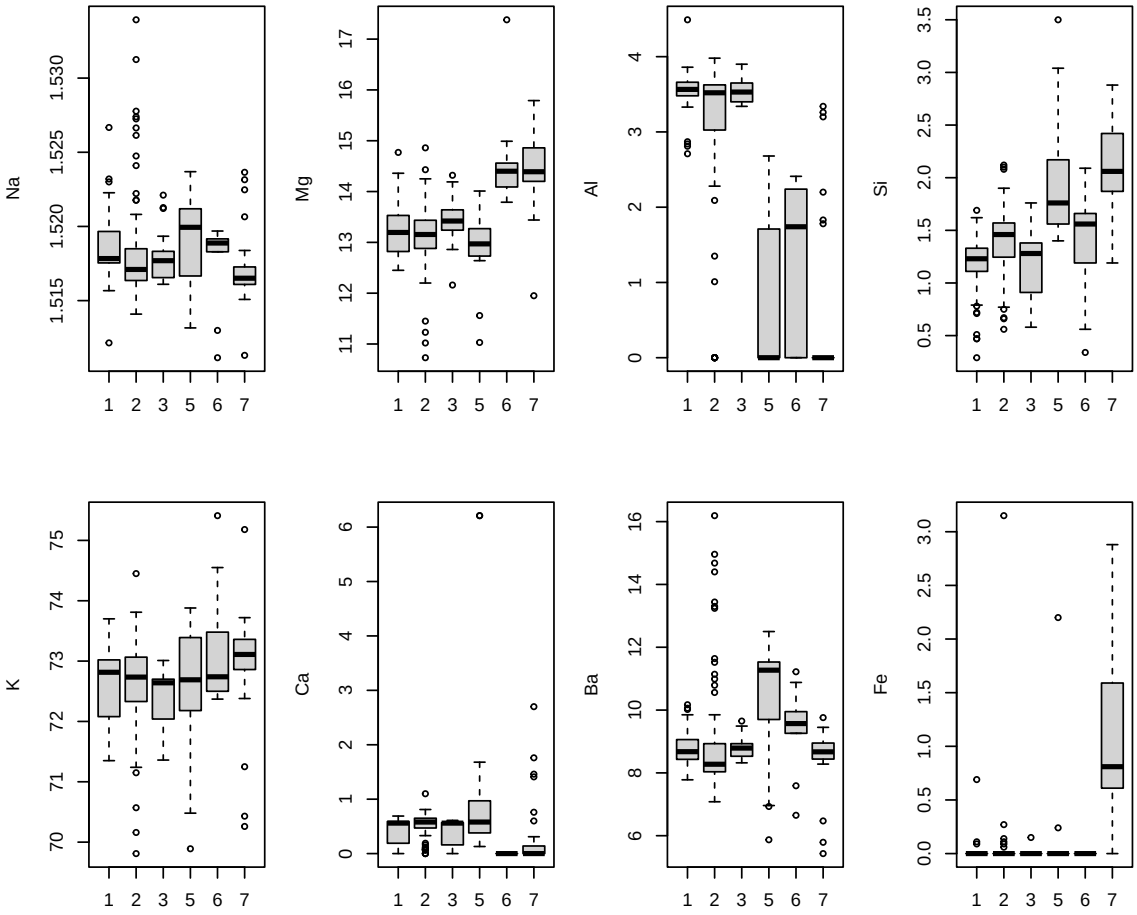
Sono inoltre riportati una serie di boxplot riferiti ai valori assunti dai diversi elementi in relazione alla tipologia di vetro (finestra o meno). In tutti i casi si può notare come varianza e mediana dei diversi elementi risultino non costanti a seconda del valore dell'attributo 'Window', segno che in un possibile modello che li leghi a 'Type' molti di essi possano essere rilevanti come variabili esplicative.

```
par(mfrow=c(2,4),mar=c(2,4.2,4,1))
names=c("Na","Mg","Al","Si","K","Ca","Ba","Fe")
for(i in 1:length(names)){
  boxplot(glass[,i]~glass$Window,xlab="Window",ylab=names[i])
}
```



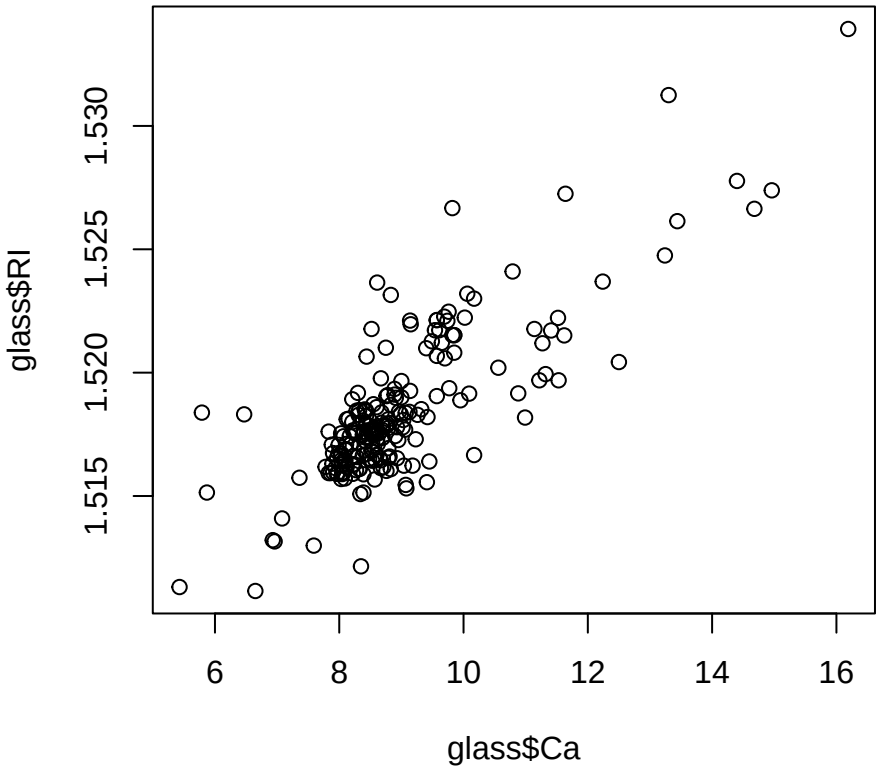
Di seguito sono presentati gli stessi boxplot precedenti ma con una granularità più fine, in questo caso i valori degli elementi nei campioni di vetro sono condizionati al livello del fattore 'Type'. Anche in questo caso si giunge alle conclusioni osservate precedentemente, cioè che considerando i diversi livelli di 'Type', la distribuzione degli elementi risulti (anche sensibilmente per certi materiali) diversa.

```
par(mfrow=c(2,4),mar=c(2,4.2,4,1))
for(i in 1:length(names)){
  boxplot(glass[,i]~glass$Type,xlab="Type",ylab=names[i])
}
```



E' stata osservata inoltre, tramite il comando pairs una possibile relazione lineare tra i valori dell'attributo 'Ca' e l'indice di rifrazione. Tale osservazione è stata supportata anche dal test di correlazione basato sul coefficiente di Pearson.

```
plot(glass$RI~glass$Ca)
```




```
cor.test(glass$RI,glass$Ca)

##
##  Pearson's product-moment correlation
##
## data:  glass$RI and glass$Ca
## t = 20.14, df = 212, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.7587539 0.8519249
## sample estimates:
##       cor
## 0.8104027
```

L'analisi esplorativa potrebbe ulteriormente proseguire con l'analisi delle componenti principali (PCA). Questo può essere svolto in quanto i possibili regressori per un modello logisitico che classifichi il livello del fattore 'Window' di nuove osservazioni possono essere molteplici.

4.3 Applicazione della libreria

Facendo riferimento al dataset introdotto precedentemente andiamo ora a presentare un possibile utilizzo della libreria XGBoost. L'obiettivo in questo caso sarà quello di realizzare diversi modelli di classificazione, basati su alberi di decisione, che siano in grado di decidere con un sufficiente grado di accuratezza il valore di alcune variabili categoriali date possibili nuove osservazioni. In particolare il

codice seguente si focalizza sulla creazione di due principali modelli, uno per la classificazione di una nuova osservazione rispetto al valore della variabile categoriale binaria 'Window', mentre il secondo per una classificazione multiclasse della variabile categoriale 'Type'. E' inoltre stata utilizzata anche la cross validation per comprendere meglio le performance ottenute dai diversi modelli. Nel realizzare tali modelli sono state utilizzate le principali funzioni che la libreria mette a disposizione dell'utente presentate nella sezione precedente.

```
# carichiamo la libreria
library("xgboost")


#impostiamo un seed casuale
set.seed(349)
#estraiamo casualmente 20 sample per il training set
test.indexes = sample(1:214, 20, F)

#suddividiamo i dati in test e training set
glass.test = glass[test_indexes,]
glass.train = glass[-test_indexes,]

#siccome la funzione XGB.DMatrix non ammette vi siano fattori, questi,
#per ogni sample sono sostituiti dai rispettivi valori numerici
#le label avranno valore 0 (non-window) oppure 1 (window).
labels.bin.test = 1-(as.numeric(unlist(glass.test[11]))-1)
labels.bin.train = 1-(as.numeric(unlist(glass.train[11]))-1)

#la stessa operazione è eseguita per quanto riguarda il fattore type.
#si noti che per n classi, i valori delle label in xgboost
#devono essere compresi tra 0 e n-1
labels.test = as.numeric(unlist(glass.test[10]))-1
labels.train = as.numeric(unlist(glass.train[10]))-1

#creiamo la DMatrix per test e training set e associamo label binarie
dtest.bin = xgb.DMatrix(data.matrix(glass.test[-c(10,11)]), label=labels.bin.test)
dtrain.bin = xgb.DMatrix(data.matrix(glass.train[-c(10,11)]),
               label=labels.bin.train)

#creiamo la DMatrix per test e training set e associamo
#la label (numerica) multiclasse
dtest = xgb.DMatrix(data.matrix(glass.test[-c(10,11)]), label=labels.test)
dtrain = xgb.DMatrix(data.matrix(glass.train[-c(10,11)]),
    label=labels.train)

#inseriamo nella watchlist il training set
wl.bin = list(train=dtrain.bin)
wl = list(train=dtrain)

#creiamo la lista di parametri (per semplificare la chiamata
#delle diverse funzioni di train)
#nel primo caso verranno utilizzati alberi con profondità massima 5, con un
#learning rate di 1e-3, con l'obiettivo di applicare sulle foglie la funzione
#logistica (in quanto il modello verrà usato per classificazione binaria).
#Inoltre specifichiamo che la computazione potrà essere distribuito su due thread
par.bin = list(max_depth = 5, eta=0.001,
           nthread=2,objective='binary:logistic', metrics = "logloss")
```

```r
#Nel secondo caso (per classificazione multiclasse) due parametri cambiano, il primo
#è 'objective' nel quale specifichiamo che il task da svolgere sarà la funzione
#softmax ed il secondo è riferito alla metrica utilizzata. Inoltre, è aggiunto un
#parametro che fa riferimento al numero di classi.
par = list(max_depth = 5, eta=0.001, num_class = 6,
           nthread=2, objective='multi:softmax',  metrics = "mlogloss")

#alleniamo i modelli (sui dati corretti, ovvero quelli di train) per 3000 epoche
M.bin = xgb.train(nrounds=3000, params = par.bin, watchlist=wl.bin,
        data = dtrain.bin)
M = xgb.train(nrounds=3000, params = par, watchlist=wl, data = dtrain)

#Utilizziamo la cross validation per meglio misurare le performance
#(avendo a disposizione un numero non troppo elevato di dati)
M.bin.cv = xgb.cv(nrounds=3000, nfold= 5, params = par.bin, data = dtrain.bin)
M.cv = xgb.cv(nrounds=3000, nfold= 5, params = par, data = dtrain)

#stampiamo graficamente le performance ottenute per quanto riguarda la CV
par(mfrow = c(2,1))
#stampiamo il grafico relativo al valore della logloss ottenuta ad ogni iterazione
#sia per il training set che per il test set
plot(train_logloss_mean~iter , data = M.bin.cv$evaluation_log, type="l",col="blue",
     ylab="error", xlab="iterations",
     ylim=range(c(train_logloss_mean, train_logloss_mean)))
lines(test_logloss_mean~iter , data = M.bin.cv$evaluation_log,col="red")
#aggiungiamo una legenda
legend("topright",legend=c("train_logloss", "test_logloss"),
       col=c("blue", "red"), lty=c(1), cex=0.8,text.font=4, bg='lightblue')

#ripetiamo le stesse operazioni ma nel caso multiclasse
plot(train_mlogloss_mean~iter , data = M.cv$evaluation_log, type="l",col="blue",
     ylab="error", xlab="iterations",
     ylim=range(c(train_mlogloss_mean, train_mlogloss_mean)))
lines(test_mlogloss_mean~iter , data = M.cv$evaluation_log,col="red")
legend("topright",legend=c("train_mlogloss", "test_mlogloss"),
       col=c("blue", "red"), lty=c(1), cex=0.8,text.font=4, bg='lightblue')
```

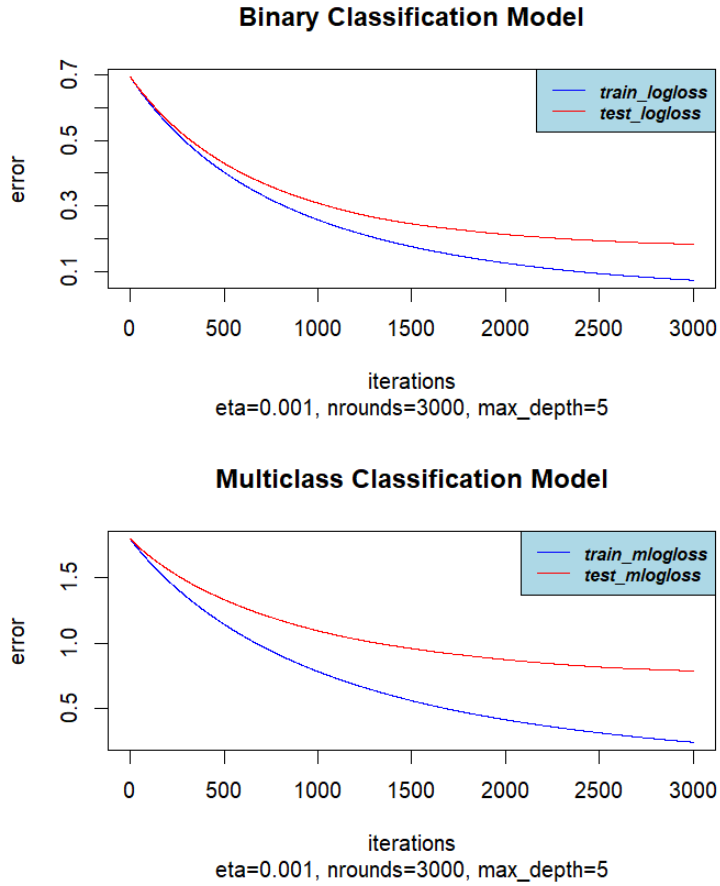


Figura 6: Performance calcolate con la cross validation nei due modelli di classificazione utilizzati

Consideriamo ora i due modelli ottenuti, il primo riferito al classificatore binario, il secondo a quello multiclasse. Sono di seguito riportati alcuni comandi R ed i rispettivi risultati ottenuti per la valutazione dei modelli considerati. In particolare utilizzeremo i modelli per prevedere i valori dei fattori dei sample del test set e calcoleremo le performance ottenute. Faremo 3 tipologie di test:

- Nel primo caso utilizzeremo il classificatore binario ottenuto per predirre il livello del fattore binario 'Window'.
- Nel secondo caso il classificatore multiclasse ottenuto per predirre sempre il livello del fattore binario 'Window' (partizionando dunque l'output del modello nelle due classi).
- Infine impiegheremo il classificatore multiclasse per predirre il livello del fattore (non binario) 'Type'.

Le predizioni saranno fatte considerando le prime 1700 (nel caso binario) e 1500 (nel caso multiclasse) iterazioni, questa scelta deriva dall'analisi del plot riportato in figura 6. Dai due grafici infatti si evince che verso 1700 e 1500 rispettivamente le curve relative agli errori sui dati di test (della cross validation) risultano diminuire la proprio inclinazione (i.e. il termine d'errore sul test risulta decrescere sempre meno). E' stata effettuata una scelta leggermente più conservativa per evitare il fenomeno dell'overfitting.

Per completezza, abbiamo effettuato i test anche considerando rispettivamente 2200 e 2000 iterazioni e abbiamo notato che le predizioni, nel nostro test set, non migliorano e le matrici di confusione sono identiche a quelle mostrate di seguito. I risultati ottenuti verranno maggiormente approfonditi nella sezione successiva.

```r
library(caret)
#creiamo un vettore con 1 e 1700, ed un secondo con 1 e 1500
#tali vettori memorizzano l'intervallo di iterazioni dei modelli
#che useremo in fase di predizione
ir.bin = c(1,1700)
ir = c(1,1500)

#salviamo i valori predetti dal primo modello (binario) sul dataset di test  per la
#variaibile binaria (Window)
predicted.bin = as.numeric(predict(M.bin, dtest, iterationrange = ir.bin)>0.5)
#stampiamo la matrice di confusione
caret::confusionMatrix(factor(labels.bin.test, levels=c(0,1)),
                       factor(predicted.bin, levels=c(0,1)))
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0  4  1
##          1  2 13
##
##                Accuracy : 0.85
##                  95% CI : (0.6211, 0.9679)
##     No Information Rate : 0.7
##     P-Value [Acc > NIR] : 0.1071
##
##                   Kappa : 0.625
##
##  Mcnemar's Test P-Value : 1.0000
##
##             Sensitivity : 0.6667
##             Specificity : 0.9286
##          Pos Pred Value : 0.8000
##          Neg Pred Value : 0.8667
##              Prevalence : 0.3000
##          Detection Rate : 0.2000
##    Detection Prevalence : 0.2500
##       Balanced Accuracy : 0.7976
##
##        'Positive' Class : 0

#salviamo i valori predetti dal secondo modello (multiclasse) sul dataset di test
#ma sempre per la variaibile binaria (Window)
predicted.bin.2 = as.numeric((predict(M, dtest, iterationrange = ir)+1)<=4)
#stampiamo la matrice di confusione relativa
caret::confusionMatrix(factor(labels.bin.test, levels=c(0,1)),
     factor(predicted.bin.2, levels=c(0,1)))
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##        0  3  2
##        1  2 13
##
##                Accuracy : 0.8
```

```
##                  95% CI : (0.5634, 0.9427)
##     No Information Rate : 0.75
##     P-Value [Acc > NIR] : 0.4148
##
##                   Kappa : 0.4667
##
##  Mcnemar's Test P-Value : 1.0000
##
##             Sensitivity : 0.6000
##             Specificity : 0.8667
##          Pos Pred Value : 0.6000
##          Neg Pred Value : 0.8667
##              Prevalence : 0.2500
##          Detection Rate : 0.1500
##    Detection Prevalence : 0.2500
##       Balanced Accuracy : 0.7333
##
##        'Positive' Class : 0
```

```
#salviamo i valori predetti dal secondo modello (multiclasse) sul dataset di test
#per quanto riguarda le singole classi predette
predicted = as.numeric(predict(M, dtest, iterationrange = ir)+1)
#stampiamo la matrice di confusione
caret::confusionMatrix(factor(labels.test+1, levels=c(1,2,3,5,6,7)),
     factor(predicted, levels=c(1,2,3,5,6,7)))
## Confusion Matrix and Statistics
##
##           Reference
## Prediction 1 2 3 5 6 7
##          1 6 1 0 0 0 0
##          2 0 2 0 1 0 0
##          3 1 0 2 1 0 0
##          5 0 0 0 1 0 0
##          6 0 0 0 0 2 0
##          7 0 0 0 0 0 0
##
## Overall Statistics
##
##                Accuracy : 0.7647
##                  95% CI : (0.501, 0.9319)
##     No Information Rate : 0.4118
##     P-Value [Acc > NIR] : 0.00343
##
##                   Kappa : 0.6852
##
##  Mcnemar's Test P-Value : NA
##
##  Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3 Class: 5 Class: 6 Class: 7
## Sensitivity            0.8571   0.6667   1.0000  0.33333   1.0000       NA
## Specificity            0.9000   0.9286   0.8667  1.00000   1.0000        1
## Pos Pred Value         0.8571   0.6667   0.5000  1.00000   1.0000       NA
## Neg Pred Value         0.9000   0.9286   1.0000  0.87500   1.0000       NA
```

```
## Prevalence             0.4118   0.1765   0.1176  0.17647   0.1176        0
## Detection Rate         0.3529   0.1176   0.1176  0.05882   0.1176        0
## Detection Prevalence   0.4118   0.1765   0.2353  0.05882   0.1176        0
## Balanced Accuracy      0.8786   0.7976   0.9333  0.66667   1.0000       NA
```

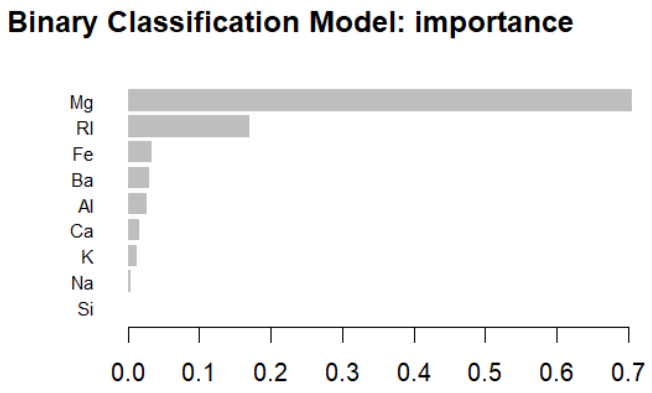
```
importance.bin = xgb.importance(feature_names = colnames(dtrain), model = M.bin)
xgb.plot.importance(importance_matrix = importance.bin)

importance = xgb.importance(feature_names = colnames(dtrain), model = M)
xgb.plot.importance(importance_matrix = importance)

xgb.ggplot.deepness(M.bin)
xgb.ggplot.deepness(M)
```

4.4 Presentazione risultati

In questa sezione presentiamo i risultati dei modelli ottenuti nella sezione precedente.



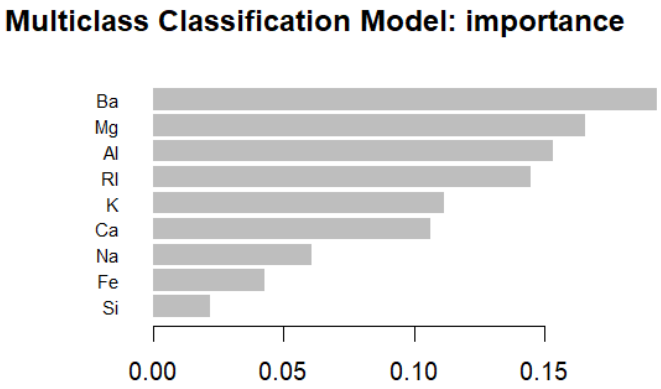


Figura 7: Grafico dell'importazza delle features nei due modelli di classificazione utilizzati

Da questi due primi grafici è possibile intuire come le variabili considerate abbiano un'importanza significativamente diversa nel caso dei due modelli considerati. In particolar modo per il modello di classificazione binario si evince il ruolo fondamentale che la variabile 'Mg' ha nello spiegare il valore di 'Window', tutte le altre variabili, ad eccezione di 'RI' in questo caso hanno un ruolo molto marginale. Tale situazione non si ripresenta però nel caso del modello multiclasse, nel quale è corretto affermare che la variabile 'Mg' sia importante nello spiegare il tipo del frammento, ma non è la sola variabile importante e nemmeno quella maggiormente rilevante. E' dunque intuibile che i modelli risultino significativamente diversi anche solamente per come le diverse variabili vengano utilizzate per spiegare la varianza della risposta.

Nei quattro grafici successivi (ottenuti tramite l'applicazione della funzione `xgb.ggplot.deepness`) sono illustrati alcuni aspetti legati alle complessità dei due modelli considerati. Per ogni classificatore infatti consideriamo il numero di foglie suddiviso per la relativa profondità alla quale si trovano e per ogni foglia un'indicazione di quanti valori osservati ricadano in essa. Com'è possibile notare nel secondo modello il numero di foglie a profondità 6 risulta significativamente maggiore rispetto al numero di foglie a parità di profondità nel primo modello. Nel modello per la classificazione binaria infatti è possibile notare come la maggioranza delle foglie si collochi a profondità 4, probabile indicatore che possa essere ragionevole una semplificazione del modello scelto. Per quanto riguarda la distribuzione dei sample un numero significativo in entrambi i modelli viene raccolto da foglie a profondità due, inoltre nel caso del modello multiclasse tale valore risulta significativamente maggiore rispetto a quello di tutti gli altri livelli.

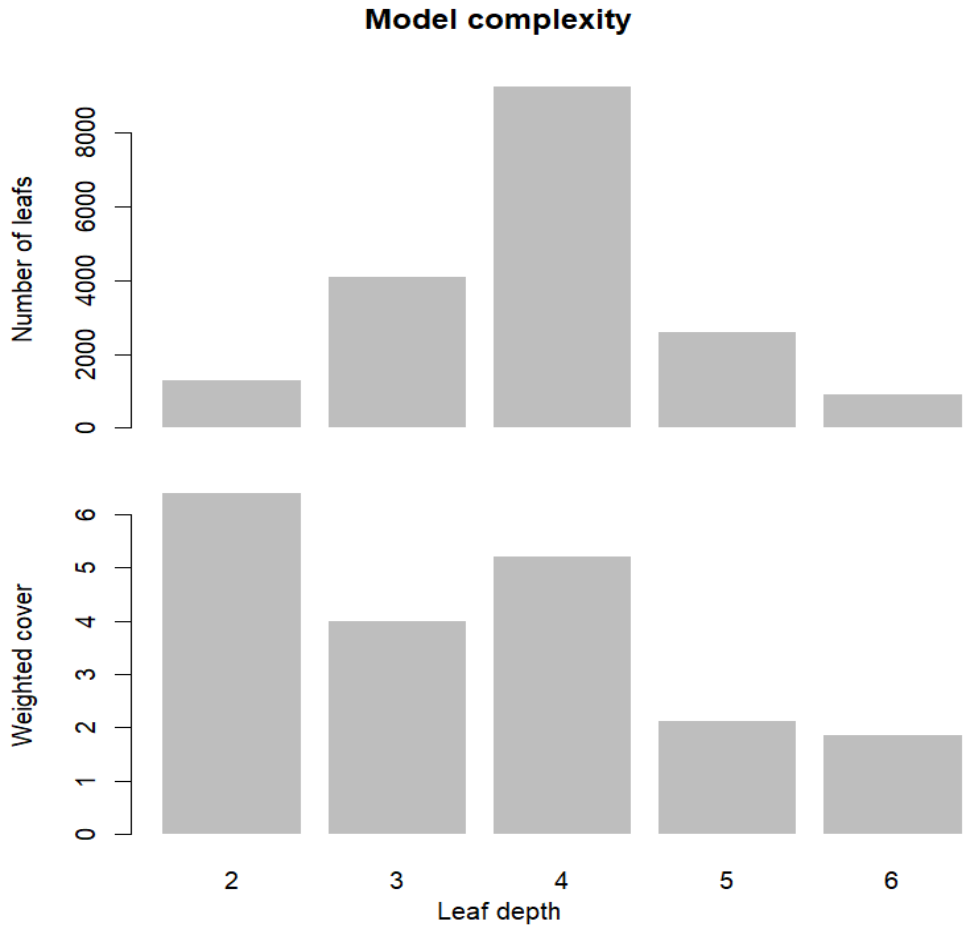


Figura 8: Grafico sulla profondità delle foglie nel modello binario

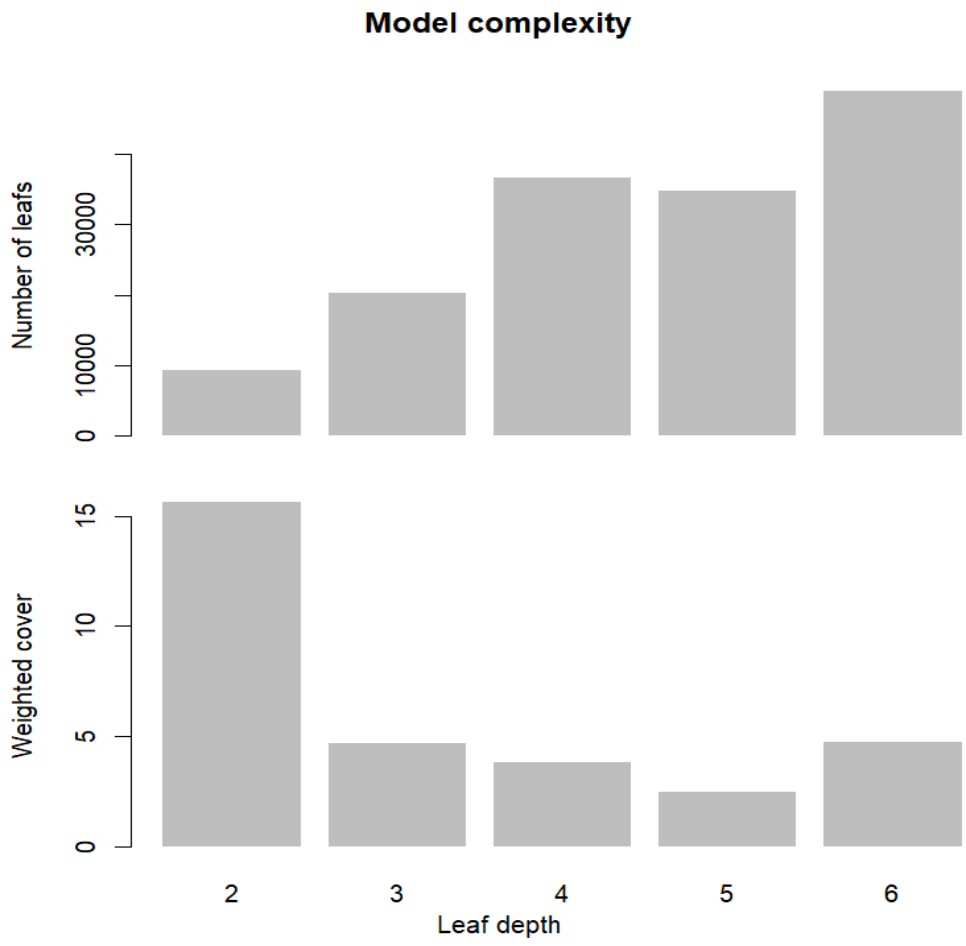


Figura 9: Grafico sulla profondità delle foglie nel modello multiclasse

4.5 Approfondimento su `xgb.create.features`

A partire dai modelli precedentemente allenati `M.bin` e `M` (con rispettivamente 1700 e 1500 iterazioni), proviamo ad applicare la funzione `xgb.create.features` per ottenere due nuovi modelli `M.bin.aug` e `M.aug` che sperabilmente miglioreranno le performance ottenute precedentemente.

```
# imposto elevata dimensione dello stack
options(expressions = 5e5)
# creo (in formato dataframe) i datataset aumentati con la funzione
# xgb.create.features sia nel caso di features aggiunte a partire dal
# classificatore binario sia nel caso del caso del classificatore multiclasse
glass.aug.bin = xgb.create.features(model = M.bin, data.matrix(glass[-c(10,11)]))
glass.aug = xgb.create.features(model = M, data.matrix(glass[-c(10,11)]))

# estraggo gli esempi di training e di test dei 2 nuovi dataset aumentati
set.seed(349)
test.indexes = sample(1:214, 20, F)
glass.test.aug.bin = glass.aug.bin[test_indexes,]
glass.train.aug.bin = glass.aug.bin[-test_indexes,]
glass.test.aug = glass.aug[test_indexes,]
```

```
glass.train.aug = glass.aug[-test_indexes,]

# creo le DMatrix di test e training del nuovo dataset
# aumentato a partire dal classificatore binario
dtrain.aug.bin = xgb.DMatrix(data = glass.train.aug.bin, label=labels.bin.train)
dtest.aug.bin = xgb.DMatrix(data = glass.test.aug.bin, label=labels.bin.test)

# creo le DMatrix di test e training del nuovo dataset
# aumentato a partire dal classificatore multiclasse
dtrain.aug = xgb.DMatrix(data = glass.train.aug, label=labels.train)
dtest.aug = xgb.DMatrix(data = glass.test.aug, label=labels.test)

# alleno due nuovi modelli M.bin.aug e M.aug rispettivamente
# classificatori binario e multiclasse che lavorano
# sui nuovi dataset aumentati a partire dai modelli precedenti
# M.bin e M
M.bin.aug = xgb.train(nrounds=1700, params = par.bin, watchlist=wl.bin,
            data = dtrain.aug.bin)
M.aug = xgb.train(nrounds=1500, params = par, watchlist=wl, data = dtrain.aug)

# controllo le performance dei nuovi modelli
library(caret)

predicted.bin.aug= as.numeric(predict(M.bin.aug, dtest.aug.bin)>0.5)
caret::confusionMatrix(factor(labels.bin.test, levels=c(0,1)),
factor(predicted.bin.aug, levels=c(0,1)))
#performace invariate a quanto con predicted.bin

predicted.bin.2.aug = as.numeric((predict(M.aug, dtest.aug)+1)<=4)
caret::confusionMatrix(factor(labels.bin.test, levels=c(0,1)),
factor(predicted.bin.2.aug, levels=c(0,1)))
# performance invariate rispetto a quanto con predicted.bin.2

predicted.aug = as.numeric(predict(M.aug, dtest.aug)+1)
caret::confusionMatrix(factor(labels.test+1, levels=c(1,2,3,5,6,7)),
factor(predicted.aug, levels=c(1,2,3,5,6,7)))
## Confusion Matrix and Statistics
##
##           Reference
## Prediction 1 2 3 5 6 7
##          1 5 1 1 0 0 0
##          2 0 3 0 1 0 0
##          3 1 0 2 1 0 0
##          5 0 0 0 1 0 0
##          6 0 0 0 0 2 0
##          7 0 0 0 0 0 0
##
## Overall Statistics
##
##                Accuracy : 0.7222
##                  95% CI : (0.4652, 0.9031)
##     No Information Rate : 0.3333
##     P-Value [Acc > NIR] : 0.0008526
##
```

```
##                  Kappa : 0.6356
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3 Class: 5 Class: 6 Class: 7
## Sensitivity            0.8333   0.7500   0.6667  0.33333   1.0000       NA
## Specificity            0.8333   0.9286   0.8667  1.00000   1.0000        1
## Pos Pred Value         0.7143   0.7500   0.5000  1.00000   1.0000       NA
## Neg Pred Value         0.9091   0.9286   0.9286  0.88235   1.0000       NA
## Prevalence             0.3333   0.2222   0.1667  0.16667   0.1111        0
## Detection Rate         0.2778   0.1667   0.1111  0.05556   0.1111        0
## Detection Prevalence   0.3889   0.2222   0.2222  0.05556   0.1111        0
## Balanced Accuracy      0.8333   0.8393   0.7667  0.66667   1.0000       NA
```

Confrontando i dati rispetto all'analisi precedente, nel nostro caso non abbiamo ottenuto vantaggi da questi due nuovi modelli. Nel caso di classificazione binaria, le performance sono rimaste invariate; nel caso invece di classificazione multiclasse le performnce sono leggermente degradate. Tutto questo con dimensioni del dataset molto più onerose, infatti `dtrain.aug.bin` presenta 8636 feature e `dtrain.aug` ha ben 68192 feature. Dunque funzione `xgb.create.features` potrebbe risultare molto utile qualora il numero di sample di un dataset di partenza sia elevato ed il numero di feature no, ma nel nostro caso applicativo non abbiamo misurato nessun incremento di performance.

4.6 Approfondimento sul multithreading

XGBoost è nota per essere molto efficiente e scalabile, con particolare attenzione verso il multithreading. Come obiettivo in questa sezione, cerchiamo di stabilire come varino le performance dell'allenamento dei modelli al cambiare del parametro `nthread`. I test che mostreremo sono stati eseguiti su un PC con 16GB di memoria ram e CPU *11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz* che dispone di 8 thread logici ma solo 4 core fisici. Per i test utilizziamo la libreria `microbenchmark` che ci permette di misurare con ragionevole precisione il tempo utilizzato per il calcolo di una funzione.

Come secondo parametro della funzione `microbenchmark`, troviamo `times`, che specifica il numero di volte che la funzione (passata come primo parametro) dovrà essere eseguita. Per un valore più stabile considereremo poi la media dei tempi delle `times` esecuzioni.

Caso 1: dataset piccolo. Consideriamo le performance di training sul dataset `glass`. Alleniamo il modello come riportato nel seguente frammento di codice.

```
# carico la libreria per effettuare benchmark
library('microbenchmark')

# creo la funzione su cui andremo a misurare le performance.
# dati in input il numero di thread e il numero di round, la funzione
# allena il modello con i parametri selezionati
train_function <- function(th, rnd){
  par = list(max_depth = 5, eta=0.001, num_class = 6,
             nthread=th, objective='multi:softmax')
  M = xgb.train(nrounds=rnd, params = par, data = dtrain)
}

# consideriamo le performance al variare dei thread tra 1,2,3,4,5,6,7,8,10,15
# e al variare delle iterazioni tra 10, 100, 500
```

```
times = list()
threads = c(1:8,10,15)
rounds = c(10,100,500)
for (j in rounds) {
  times[[j]] = vector()
  for(i in threads){
    # misuriamo i tempi della funzione di training del modello
    # per 5 volte. prendiamo in considerazione il tempo medio
    bench = microbenchmark(train_function(i,j), times=5)
    times[[j]] = c(times[[j]], mean(bench$time))
  }
}


# stampiamo in unico grafico le performance ottenute
# sulle x troviamo il numero di thread e sulle y il tempo di esecuzione.
# per avere i tempi in ms, dobbiamo divedere i tempi misurati per un fattore
# di scala pari a 1000000
scale = 1000000
par(mfrow = c(1,3))
plot(threads, times[[10]]/scale, ylab="time (ms)", xlab = "nthread", type = "o",

col = "red", main = "rounds=10")
plot(threads, times[[100]]/scale, ylab="time (ms)", xlab = "nthread", type = "o",
col = "blue", main = "rounds=100")
plot(threads, times[[500]]/scale, ylab="time (ms)", xlab = "nthread", type = "o",
col = "green", main = "rounds=500")
```

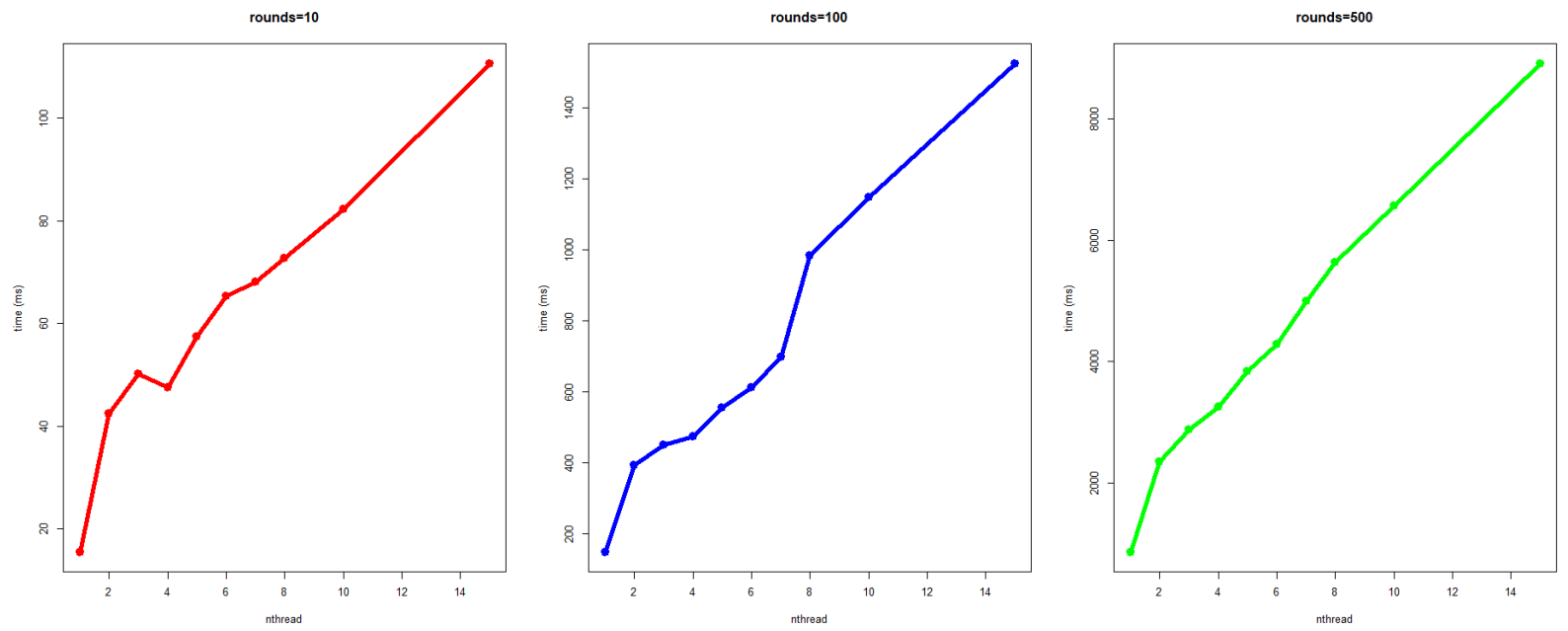


Figura 10: Grafico performance di training: tempo di allenamento confrontato con il numero di thread utilizzati. Test con `nrounds=10`, `nrounds=100` e `nrounds=500`.

Dal grafico presentato (figura 10) si nota chiaramente che per il dataset utilizzato (214 samples):

- Si hanno maggiori performance utilizzando un singolo thread.
- I thread non vengono maggiormente utilizzati all'aumentare del numero di round di allenamento.

Il grafico ci suggerisce quindi che per dataset relativamente piccoli come il `glass` sia più efficiente utilizzare un singolo thread indipendentemente dal numero di round. Questo abbiamo ipotizzato sia dovuto alla suddivisione del lavoro tra i vari thread. In questo caso è possibile che l'overhead per la creazione, sincronizzazione e gestione dei thread risulti annullare il beneficio portato dalla (breve, in termini di mole di lavoro) esecuzione concorrente.

Caso 2: dataset grande. Visti i risultati deludenti appena discussi, ci chiediamo se con dataset che presenta un maggiore numero di sample, l'aumento del numero di thread possa ridurre i tempi di allenamento. Proviamo quindi ad aumentare il nostro dataset casualmente per arrivare a 5000 sample.

```
# carichiamo le librirei
library("xgboost")
library('microbenchmark')

# vogliamo considerare un dataset di dimensione 5000
# con il fine di misurare le performance multithread

# consideriamo il glass dataset (con 214 sample)
# memorizziamo nel vettore sds le deviazioni standard delle
# varie variabili esplicative
glass.ext = glass
dim = 214
ext.dim = 5000
sds = c()
for(i in 1:9){
  sds = c(sds, sd(glass[,i]))
}

# creiamo il nuovo dataset esteso a 5000 sample 'glass.ext'
# con la seguente procedura:
#  -1 estraiamo random uno dei 214 sample originari
#  -2 ad ogni campo del sample aggiungiamo un errore casuale
#     considerando una distribuzione normale di media 0
#     e varianza, la varianza campionaria osservata
#     e misurata in precedenza
for(i in 1:(ext.dim-dim)){
  j = sample(1:dim)
  new.sample = glass.ext[j,]
  for(k in 1:9){
    new.sample[k] = new.sample[k] + rnorm(1,0,sds[k])
  }
  glass.ext[dim+i,] = new.sample

}

# abbiamo ottenuto il dataset aumentato

# creiamo le label aumentate e il dataset aumentato
# in formato xgb.DMatrix
```

```
labels.ext = as.numeric(unlist(glass.ext[10]))-1
dtrain.ext = xgb.DMatrix(data.matrix(glass.ext[-c(10,11)]), label=labels.ext)

# creiamo la funzione di cui misureremo le performance.
# prende in input il numero di thread e allena il modello
# con i parametri selezionati
train_function <- function(i){
  par = list(max_depth = 5, eta=0.001, num_class = 6,
             nthread=i, objective='multi:softmax')
  M = xgb.train(nrounds=100, params = par, data = dtrain.ext)
}

# consideriamo i tempi al variare dei thread
times = c()
threads = c(1:8,10,15)
for(i in threads){
  # misuriamo i tempi per 5 volte e consideriamo il tempo medio
  bench = microbenchmark(train_function(i), times=5)
  times = c(times, mean(bench$time))
}

# stampiamo il grafico dei risultati ottenuti
# al solito, troviamo slle y i tempi misurati e sulle x
# il numero di thred
par(mfrow = c(1,1))
scale = 1000000
plot(threads, times/scale, ylab="time (ms)", xlab = "nthread", type = "o",
col = "red", main = "rounds=100", lwd=4)
```

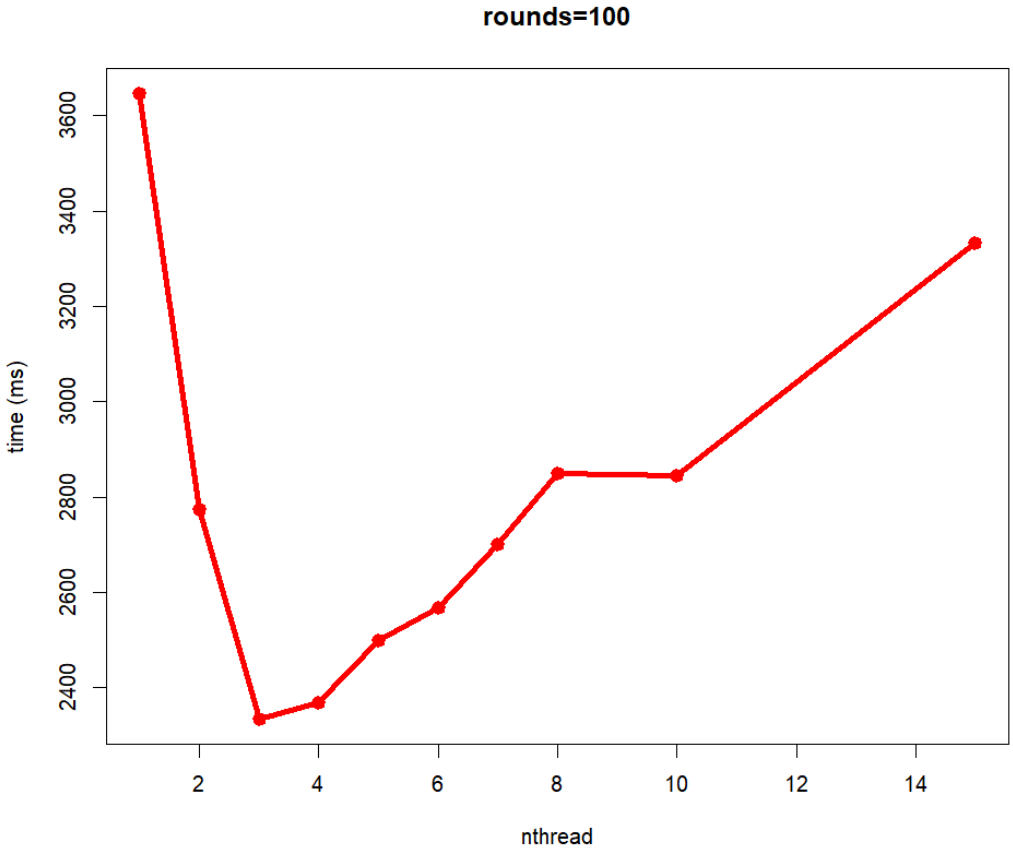


Figura 11: Grafico performance di training sul `glass` dataset aumentato a 5000 samples.: tempo di allenamento confrontato con il numero di thread utilizzati. Test con `nrounds=100`.

Dal grafico riportato sopra (figura 11) emerge che il numero di thread influenza sensibilmente le prestazioni e in particolare un singolo thread non è più la scelta ottimale. Con la configurazione utilizzata appare abbastanza evidente che impostando il lavoro su 3 o 4 thread facendo riferimento ad un dataset di relativamente grandi dimensioni (5000 sample), questa risulti essere la scelta ottimale. Si noti che la nostra configurazione dispone di soli 4 core fisici. E' dunque plausibile che su dataset con centinaia di migliaia di sample ed un elevato numero di core a disposizione i tempi per l'allenamento del modello beneficino sensibilmente della possibilità di sfruttare un elevato numero di thread.

5 Conclusioni

Nel presente lavoro abbiamo analizzato brevemente il metodo dei gradient boosting tree, sia dal punto di vista teorico che pratico. In particolare abbiamo spiegato da cosa derivano i tre termini gradient, boosting e tree ed abbiamo spiegato come venga realizzato un regressore o classificatore con questa metodologia. Abbiamo affrontato i problemi chiave relativi a questo metodo ossia la scelta del numero di iterazioni, la scelta del learning rate η e la scelta della profondità dei singoli alberi. Abbiamo considerato la libreria XGBoost che rappresenta un'implementazione state-of-the-art riguardo questa metodologia, presentandone vari aspetti. XGBoost è disponibile in vari linguaggi di programmazione tra cui R e python e pone particolare attenzione all'efficienza utilizzando specifiche implementazioni di algoritmi greedy, algoritmi parallelizzabili su più thread e gestione efficiente della cache. Le funzioni della libreria si adattano a molti utilizzi: E' infatti possibile utilizzare XGBoost in maniera semplice con pochi comandi e poche linee di codice, oppure è possibile un utilizzo più profondo,

personalizzando i vari parametri e le varie callback (eventualmente definendone di nuove) in modo da ottenere risultati mirati. Abbiamo presentato le varie funzioni principali (a nostro avviso) mettendo in luce alcuni casi tipici in cui esse potrebbero essere utilizzate. Abbiamo testato la libreria sul glass datset dalla repository UCI e considerando la piccola quantità di sample a disposizione e lo sbilanciamento tra le classi, riteniamo di aver ottenuto risultati soddisfacenti. Abbiamo testato le performance di multithreading notando che per piccoli dataset risulti più efficiente usare un singolo thread, mentre con dataset di medie grandi dimensioni è conveniente usare più thread anche a seconda della configurazione di cui si dispone.

In conclusione, il metodo dei gradient boosting tree è sicuramente da considerare quando si parla di classificazione e regressione, e la libreria XGBoost permette di implementare velocemente (ed efficientemente) i vari modelli.

5.1 Possibili sviluppi futuri

Tra i possibili sviluppi futuri vi è quello di un confronto tra gli alberi di decisione ottenuti con il gradient boosting ed altri modelli (di classificazione) come ad esempio la regressione logistica. In particolare, considerando il dataset presentato si potrebbe utlizzare la PCA in modo da poter realizzare diversi modelli ed infine confrontarli. Un altro possibile sviluppo maggiormente legato alla libreria potrebbe essere quello di testare le performance di alcuni metodi (e.g. il metodo per costruire l'albero) con dataset che presentino un elevato numero di feature, per osservare se le performance dei metodi utilizzati con più thread scalino in maniera corretta anche in base al numero di feature e non solo rispetto a quello di sample del dataset.

Riferimenti bibliografici

[Athitsos and Sclaroff, 2005] Athitsos, V. and Sclaroff, S. (2005). Boosting nearest neighbor classifiers for multiclass recognition. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)-Workshops*, pages 45–45. IEEE.

[Bioch et al., 1997] Bioch, J. C., Meer, O. v. d., and Potharst, R. (1997). Bivariate decision trees. In *European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 232–242. Springer.

[Chen et al., 2019] Chen, T., He, T., Benesty, M., and Khotilovich, V. (2019). Package 'xgboost'. *R version*, 90:1–66.

[Evett and Spiehler, 1989] Evett, I. W. and Spiehler, E. J. (1989). Rule induction in forensic science. In *Knowledge Based Systems*, pages 152–160.

[He et al., 2014] He, X., Pan, J., Jin, O., Xu, T., Liu, B., Xu, T., Shi, Y., Atallah, A., Herbrich, R., Bowers, S., et al. (2014). Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the eighth international workshop on data mining for online advertising*, pages 1–9.

[Natekin and Knoll, 2013] Natekin, A. and Knoll, A. (2013). Gradient boosting machines, a tutorial. *Frontiers in Neurorobotics*, 7.