ALBERI DI REGRESSIONE

Claudio Poli

1.INTRODUZIONE

L'**apprendimento supervisionato** è una tecnica che pone le sue basi su istanze associate ad un vettore di features, le quali vengono considerate per definire variabili di input utili alla previsione di una o più variabili di output. Queste ultime possono essere di due tipi: quantitative (problemi di regressione) e qualitative (problemi di classificazione). Prendendo in considerazione problemi di **regressione** si indica con $Y = \gamma \subset \mathbb{R}$ la variabile di output ed occorre determinare una funzione **previsore ottimale** $\phi(X)$ con $\phi: \chi \to \gamma$ per prevedere Y dato $X \in \chi$, per fare ciò è necessario ricorrere alla teoria statistica delle decisioni. In primis è necessario definire una funzione di perdita $L: \gamma \times \chi \to [0, +\infty]$ e, solitamente, la scelta ricade sulla perdita quadratica $L[Y, \phi(X)] = [Y - \phi(X)]^2$, la quale deve essere minimizzata dal previsore ϕ, che equivale a minimizzare l'EPE(Expected squared Prediction Error):

$$EPE(\phi) = E[Y - \phi(X)]^2 = \int [y - \phi(x)]^2 f(x, y) dxdy$$

dove $f(x, y)$ indica la distribuzione di probabilità congiunta a $(X, Y) \in \mathbb{R}^{p+1}$. La minimizzazione dell'errore dunque equivale a $argmin_{\phi} E[Y - \phi(X)]^2$, per cui applicando la legge del valore atteso condizionale iterato $h(X, Y) = [Y - \phi(X)]^2$ e, aggiungendo e sottraendo $E_{Y|X}(Y|X)$ si otterrà $E[Y - \phi(X)]^2 = E_X E_{Y|X}\{[Y - \phi(X)]^2|X\} = E_X E_{Y|X}\{[Y - E_{Y|X}(Y|X) + E_{Y|X}(Y|X) - f(X)]^2|X\}$.
Sviluppando il quadrato il primo termine corrisponderà a $E_{Y|X}\{[Y - E_{Y|X}(Y|X)]^2|X\} = Var_{Y|X}(Y|X)$, il secondo sarà $E_{Y|X}\{[E_{Y|X}(Y|X) - \phi(X)]^2|X\}$ ed il termine misto avrà espressione risultante $2E_{Y|X}\{[Y - E_{Y|X}(Y|X)][E_{Y|X}(Y|X) - \phi(X)]|X\} = 2[E_{Y|X}(Y|X) - \phi(X)] \times E_{Y|X}\{[Y - E_{Y|X}(Y|X)]|X\}$. Quest'ultimo sarà nullo in quanto $E_{Y|X}\{[Y - E_{Y|X}(Y|X)]|X\} = E_{Y|X}(Y|X) - E_{Y|X}(Y|X) = 0$. L'errore di previsione sarà dunque $E[Y - \phi(X)]^2 = Var_{Y|X}(Y|X) + E_X E_{Y|X}\{[E_{Y|X}(Y|X) - \phi(X)]^2|X\}$ e, dato che $[E_{Y|X}(Y|X) - \phi(X)]^2 \geq 0$ si avrà che

$$\phi(X) = E_{Y|X}(Y|X = x)$$

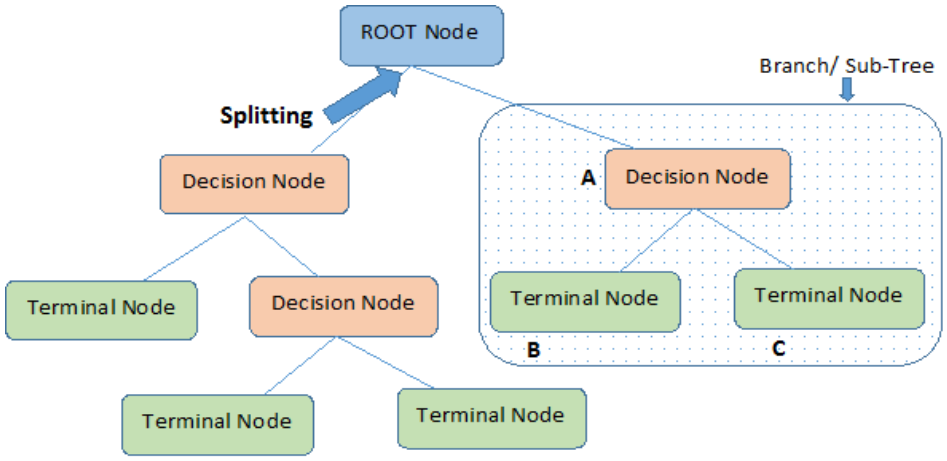
Il valore atteso condizionale è considerato il previsore ottimale dell'output ed è noto come **curva di regressione**, mentre funzione di predita quadratica è nota invece come perdita L_2.

1.1 ALBERI DI DECISIONE

Gli alberi di decisione rappresentano una tipologia di apprendimento supervisionato e vengono utilizzati sia in problemi di regressione che di classificazione e possono essere di due tipi:

- **Alberi di decisione a variabili categoriche**: utili a prevedere variabili di output discrete, codificando l'appartenenza di un istanza a due o più classi;
- **Alberi di decisione a variabili continue**: utili a prevedere variabili di tipo quantitativo che possono assumere tutti i possibili valori in un dato intervallo.

1.1.1 Terminologia



Tramite l'immagine è possibile identificare differenti elementi:

- La **radice** rappresenta l'intera popolazione e verrà suddivisa in due o più insiemi omogenei
- Lo **splitting** costituisce il processo di divisione di un nodo in due o più sotto nodi
- Il **nodo di decisione** rappresenta il sotto nodo che viene a sua volta suddiviso in sotto nodi
- Il **nodo foglia/terminale** costituisce un nodo che non viene suddiviso ulteriormente
- Il **pruning** è il processo di rimozione di sotto nodi da un nodo di decisione e rappresenta l'opposto dello splitting
- Un **branch** rappresenta la sotto sezione di un albero

- Il **nodo padre** rappresenta il nodo che viene suddiviso in sotto nodi, chiamati a loro volta **nodi figli**

1.1.2 Vantaggi e svantaggi

- **Interpretabilità**: l'output di un albero decisionale è molto facile da comprendere e non richiede alcuna conoscenza statistica;

- **Utile per l'esplorazione dei dati**: l' albero decisionale è uno dei modi più veloci per identificare le variabili più significative e la relazione tra di esse. Con l'aiuto di questa tipologia di alberi è possibile creare nuove variabili/features con migliori capacità predittive. Inoltre può essere utilizzato anche nella fase di esplorazione dei dati, identificando le variabili più significative;

- **Minor pulizia dei dati richiesta**: è richiesta una minor pulizia dei dati rispetto ad altre tecniche di modellazione essendo poco influenzato da valori anomali o mancanti;

- **Il tipo di dato non è un vincolo**: può gestire variabili sia numeriche che categoriche;

- **Metodo non parametrico**: non possiedono ipotesi in merito allo spazio di distrubuzione e sulla struttura del classificatore;

- **Overfitting**: costituisce una delle difficoltà principali che, però, può essere risolta settando i giusti vincoli sui parametri del modello e sul pruning;

2 ALBERI DI REGRESSIONE

2.1 Struttura

Per sviluppare un albero di regressione occorre considerare p dati input e una risposta per tutte le N osservazioni: (x_i, y_i) per $i = 1,2,\dots,N$, con $x_i = (x_{i1}, x_{i2}, \dots, x_{ip})$. L'algoritmo avrà il compito di definire automaticamente le

variabili e i punti di splitting ed anche la topologia dell'albero. Supponendo di effettuare una partizione in M regioni $R_1, R_2, \ldots, R_M$ e modellare la risposta come una costante c_m in ciascuna regione:

$$f(x) = \sum_{m=1}^{M} c_m \, I(x \in R_m)$$

Adottando il criterio della minimizzazione della somma dei quadrati $sum(y_i - f(x_i))^2$ si può osservare che la miglior c_m corrisponderà alla media delle y_i nella regione R_m:

$$c_m = avg(y_i | x_i \in R_m)$$

Dal momento che risulterà impossibile da un punto di vista computazionale trovare la migliore partizione binaria in termini di somma minima di quadrati, è necessario procedere con un algoritmo greedy. Partendo dai dati completi, si consideri una variabile di divisione j e un punto di divisione s e si definisca la coppia di semipiani

$$R_1(j, s) = \{X | X_j \leq s\}; R_2(j, s) = \{X | X_j > s\}$$

Quindi si ricerca la variabile di divisione j e il punto di divisione s che risolvono

$$min_{j,s}[min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2]$$

Per ogni scelta di j e s la minimizzazione interna è risolta da

$$c_1 = avg(y_i | x_i \in R_1(j, s)) e c_2 = avg(y_i | x_i \in R_2(j, s))$$

Per ogni variabile di divisione, la determinazione del punto di divisione può essere effettuata molto rapidamente valutando tutti gli input. Una volta trovata la migliore divisione, i dati vengono partizionati nelle due regioni risultanti ed il processo viene ripetuto su ciascuna delle due regioni. Sviluppando un albero è possibile andare incontro ad alcune problematiche, con un albero molto grande si può incorrere in overfitting sui dati, mentre un albero piccolo potrebbe non catturarne la struttura.

2.2 Differenze tra alberi di regressione e alberi di classificazione

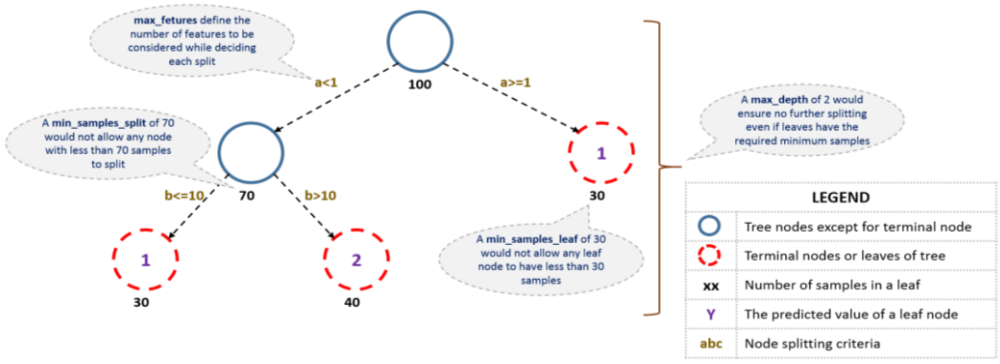
Nonostante entrambe le tipologie di alberi abbiano un funzionamento simile esistono diverse differenze tra di loro, le principali sono:

1. Gli alberi di regressione vengono utilizzati quando la variabile dipendente è continua mentre gli alberi di classificazione vengono utilizzati quando la variabile dipendente è categorica;
2. Negli alberi di regressione, il valore ottenuto dai nodi terminali nei dati di training è la risposta media delle osservazioni che ricadono in quella regione;
3. In caso di albero di classificazione, il valore (classe) ottenuto dal nodo terminale nei dati di training è la moda delle osservazioni che ricadono in quella regione;
4. Entrambi gli alberi dividono lo spazio di predizione (variabili indipendenti) in regioni distinte e non sovrapposte;
5. Entrambi gli alberi seguono un approccio greedy dall'alto verso il basso noto come *divisione binaria ricorsiva*;
6. Il processo di suddivisione prosegue finché non viene raggiunto un *criterio di arresto* definito dall'utente;
7. In entrambi i casi, il processo di divisione si traduce in alberi completamente sviluppati fino a quando non vengono raggiunti i criteri di arresto.

Un problema che affligge gli alberi di regressione è quello dell'**overfitting**, che implica una scarsa capacità predittiva su dati diversi da quelli con i quali è stata effettuata la fase di training. Per ovviare a ciò viene utilizzato il processo di **'pruning'**.

2.3 Parametri per evitare overfitting

L'overfitting è una delle principali problematiche affrontate durante l'utilizzo degli algoritmi tree-based e per prevenirlo si può procedere utilizzando due metodi:



1. **Impostazione dei vincoli sulla dimensione dell'albero**

- Numero di samples minimi per uno split
  - Definizione di un numero minimo di samples richiesti da un nodo per considerare lo splitting;
  - Valori elevati impediscono a un modello di apprendere relazioni che potrebbero risultare altamente specifiche per un particolare sample;
  - Valori bassi possono portare a under-fitting, dovrebbe essere sottoposto a tuning mediante CV.
- Numero minimo di samples per un nodo terminale (foglia)
  - Definisce il numero minimo di samples (o osservazioni) richiesti in un nodo foglia.
- Profondità massima dell'albero
  - Maggiore profondità consentirà al modello di apprendere relazioni molto specifiche per un particolare sample;
  - Sfrutta tuning con CV;
- Numero massimo di nodi terminali
  - Può essere definito al posto di max_depth. Poiché vengono creati alberi binari, una profondità di n produrrebbe un massimo di 2^n foglie;
- Numero massimo di features da considerare per lo split
  - Selezionate randomicamente;
  - Generalemente viene utilizzata la radice quadrata delle features totali;
  - Valori più alti possono portare ad overfitting.

2. **Pruning**
   E' una tecnica di machine learning che riduce la dimensione degli alberi decisionali rimuovendo le sezioni dell'albero che forniscono poca capacità di classificazione delle istanze. La potatura diminuisce la complessità del classificatore finale e ne migliora l'accuratezza predittiva riducendo l'overfitting. Questa tecnica può essere eseguita dall'alto verso il basso o viceversa, nel primo caso verranno attraversati i nodi ed i sottoalberi a partire dalla radice verranno potati, mentre nel caso in cui si proceda dal basso verso l'alto si inizierà dai nodi foglia.

Cost complexity pruning: questa tecnica genera una serie di alberi $T_0 \dots T_m$ dove T_0 è l'albero originale e T_m è la radice, ad ogni passo viene rimossa la foglia per il quale si riscontra l'errore quadratico più elevato. Si indicizzino i nodi terminali per m, con il nodo m che rappresenta la regione R_m. Sia $|T|$ il numero di nodi terminali in T. Con

$$N_m = \#\{x_i \in R_m\}$$

$$c_m = 1/N_m \sum_{x_i \in R_m} y_i$$

$$Q_m(T) = 1/N_m \sum_{x_i \in R_m} (y_i - c_m)^2$$

si definisca il criterio della complessità dei costi

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m \, Q_m(T) + \alpha|T|$$

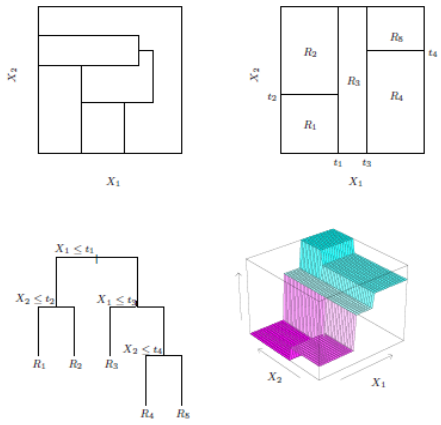
L'idea è quella di trovare, per ogni α, il sottoalbero $T_\alpha \subseteq T_0$ che minimizzi $C_\alpha(T)$. Il parametro di tuning $\alpha \geq 0$ controlla il compromesso tra la dimensione dell'albero e la sua capacità di adattamento ai dati. Valori elevati di α producono alberi più piccoli T_α, e viceversa per valori minori di α. Con $\alpha = 0$ verrà indicato l'albero completo T_0. Per scegliere in modo adattivo α è necessario dimostrare che per ogni α esiste un sottoalbero T_α unico e più piccolo che minimizza $C_\alpha(T)$. Per trovare T_α si utilizza la *weakest linking pruning* che consiste nel collassare il nodo interno che produce il più piccolo aumento per nodo in $\sum_m N_m \, Q_m(T)$ e si continua finché non si ottiene l'albero a nodo singolo (radice).

3 TREE BASED-METHODS

3.1 Cart

I metodi tree-based partizionano lo spazio delle features in sottoinsiemi e successivamente effettuano il fitting di un modello semplice su ognuno di essi. Uno dei metodi più popolari è il CART e viene utilizzato sia per problemi di regressione che di classificazione.

Prendendo in considerazione un problema di regressione con risposta continua Y e input X_1 e X_2:



Nel grafico in alto a sinistra è possibile osservare il partizionamento dello spazio delle features tramite linee parallele agli assi. In ogni elemento della partizione, Y viene modellato con una costante diversa. Tuttavia, sebbene ogni linea di partizionamento è definita semplicemente come $X1 = c$, alcune delle regioni risultanti sono complesse da descrivere. Per semplificare le cose, si può limitare l'attenzione alle partizioni binarie ricorsive come quella posta nel grafico in alto a destra. Per prima cosa è necessario dividere lo spazio in due regioni per poi modellare la risposta in base alla media di Y in ciascuna di esse, scegliendo la variabile e il punto di splitting per ottenere il miglior fit. Poi, a loro volta, una o entrambe le regioni vengono divise in altre due e questo processo viene iterato fino a quando non viene verificata una regola di arresto. Ad esempio, nella sezione posta in alto a destra, prima è stata divisa a $X_1 = t_1$, poi la regione $X1 \leq t1$ è diviso a $X2 = t2$ e la regione $X1 > t1$ a $X1 = t3$. Infine, il la regione $X1 > t3$ è divisa a $X2 = t4$. Il risultato di questo processo è una partizione in cinque regioni $R_1, R_2, \ldots, R_5$ ed il corrispondente modello di regressione predirà Y con una costante c_m nella regione R_m, ovvero

$$f(X) = \sum_{m=1}^{5} c_m \, I(X_1, X_2) \in R_m$$

Questo modello può essere rappresentato dall'albero binario posto in basso a sinistra, dove l'intero set di dati si trova nella parte superiore dell'albero. Le osservazioni che soddisfano la condizione ad ogni giunzione sono assegnati al ramo sinistro, e gli altri al ramo destro. I nodi terminali o le foglie dell'albero corrispondono alle regioni $R_1, R_2, \ldots, R_5$; questo grafico evidenzia il vantaggio degli alberi binari, ossia la loro interpretabilità. Il riquadro in basso a destra, invece, è un grafico prospettico della superficie di regressione del modello in questione. Il partizionamento dello spazio delle features è completamente descritto da un singolo albero. Con più di due input alcune partizioni sono difficili da rappresentare, ma la definizione ad albero binario funziona allo stesso modo.

3.2 Splitting

Riduzione della varianza

Questo algoritmo utilizza la formula standard della varianza:

$$Variance = \sum(X - \bar{X})^2/n$$

dove $\bar{X}$ è la media dei valori, X è il valore attuale e n è il numero dei valori. Per calcolare la varianza occorre:

1. Calcolare la varianza per ogni nodo;
2. Calcolare la varianza per ogni split come una media pesata per la varianza di ogni nodo.

Lo split con varianza minore viene scelto come criterio per dividere i dati

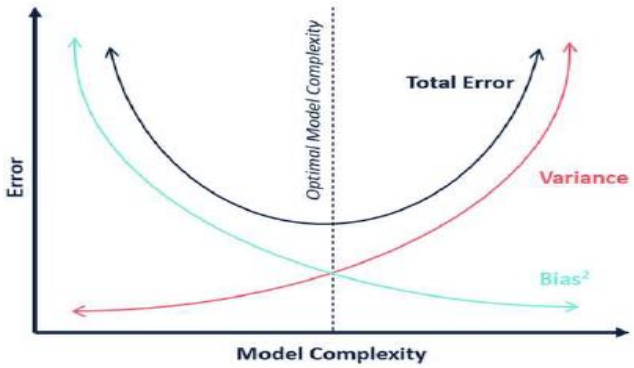
3.3 Metodi Ensemble

I metodi Ensemble coinvolgono gruppi di modelli predittivi per ottenere una migliore precisione e stabilità del modello rispetto ai modelli tree-based classici. Come ogni altro modello, anche un algoritmo tree-based soffre di problemi relativi a bias e varianza. Come già visto in precedenza, man mano che la complessità del modello aumenta, è possibile notare una riduzione dell'errore di previsione dovuto da un bias minore nel modello, al contrario invece, comincerà a farsi presente il problema dell'overfitting ed il modello soffrirà di varianza elevata.Occorre dunque bilanciare queste due componenti, gestendo il noto **trade-off bias-varianza**.

Per comprendere questa tecnica occorre considerare l'errore atteso di test del modello di regressione $Y = \phi(X) + \varepsilon$ l'errore atteso di test prende il nome di **errore quadratico medio di previsione (RMSE)**:

$$Err(x_0) = E_{\mathbb{D},Y_0}(Y_0 - \hat{Y}_0)^2$$

Per cui è necessario considerare anche il lemma per il quale Z è una variabile aleatoria con valore atteso $\bar{Z} = E(Z)$ e quindi $E(Z - \bar{Z})^2 \Longrightarrow E(Z^2) = E(Z - \bar{Z})^2) + \bar{Z}^2$. Applicando il lemma due volte è possibile arrivare alla conclusione che $Err(x_0) = \sigma^2 + Var_{\mathbb{D}}(\hat{Y}_0) + [E_{\mathbb{D}}(\hat{Y}_0) - \phi(x_0)]^2$, dove σ^2 è la variabilità dell'output attorno al valore atteso $E(Y_0) = \phi(x_0)$ e $\sigma^2 \equiv E_{Y_0}(Y_0 - \phi(x_0))^2$. eliminando la dipendenza dal training set. Questa quantità verrà identificata con errore irriducibile. Il terzo termine è il **termine di bias** che misura quanto in media la previsione $\hat{Y}_0$ differisce dal vero valore atteso. Diremo che un previsore empirico $\hat{\phi}(x_0)$ è corretto se $E_{\mathbb{D}}(\hat{Y}_0) = \phi(x_0)$, situazione che si verifica con il modello di regressione lineare per il quale il termine di bias è nullo. Nel caso contrario, ad esempio per previsori empirici come la ridge regression, il termine di distorsione è diverso da zero. Anche nel caso in cui quest'ultimo sia nullo un elemento da non tralasciare è il termine di varianza $Var_{\mathbb{D}}(\hat{Y}_0)$ il quale misura l'erraticità della previsione attorno al proprio valore atteso, al variare del training set. Per questo viene definito il teorema di decomposizione bias-varianza che è valido per qualsiasi modello di regressione lineare con previsore $\phi(x)$: $Err(x_0) = \sigma^2 + Var_{\mathbb{D}}(\hat{Y}_0) + [E_{\mathbb{D}}(\hat{Y}_0) - \phi(x_0)]^2$. Grazie a questo teorema è possibile osservare che all'aumentare della complessità del modello, questo si adatterà meglio ai dati di training ed il termine di distorsione decrescerà, mentre aumenterà quello della varianza. Dal momento che l'interesse primario è quello della previsione su istanze future è necessario minimizzare l'errore atteso di test. Dato che che stimare l'errore di generalizzazione risulta complesso, una possibilità è quella di calcolare la perdita media, nota come rischio empirico $\bar{err} = (1/N)\sum_{i=1}^{N} L\,(y_i, \hat{\phi}(x_i))$ ed effettuando quest'operazione è possibile incorrere in fenomeni quali underfitting o overfitting. Per questo motivo è necessario definire un punto di ottimo atto a minimizzare l'errore atteso di test e questo si effettua tramite le fasi di model selection e inferenziale (con relativa fase di tuning dei parametri).



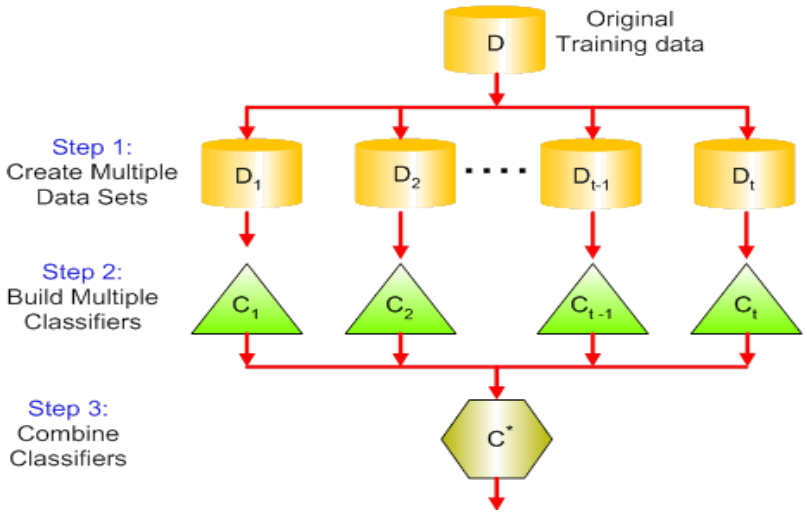
3.3.1 Bagging

Il bagging (Bootstrap Aggregation) è una tecnica utilizzata per ridurre la varianza delle previsioni combinando il risultato di più classificatori modellati su diversi sottocampioni dello stesso dataset.

$$f_{bag}(x) = 1/B \sum_{b=1}^{B} f_b\,(x)$$

in cui si generano B diversi dataset di training bootstrap.
Successivamente viene effettuato il training del modello sul b-esimo set di training bootstrap per ottenere $f_b(x)$ e infine si effettua la media delle previsioni.

Di seguito si evidenziano i tre step del bagging:



1. **Creazione di dataset multipli**
   Il sampling viene effettuato con la sostituzione dei dati originali creando nuovi dataset, i quali possono essere formati da una frazione delle colonne e delle righe.

2. **Creazione di più classificatori** su ogni set di dati. In generale, è possibile utilizzare lo stesso classificatore per creare modelli e previsioni.

3. **Combinazione delle predizioni dei classificatori** tramite l'utilizzo di media, moda o mediana, a seconda del problema. In genere, questi valori combinati sono più robusti di un singolo modello.

Per applicare il bagging agli alberi di regressione è sufficiente costruire B alberi utilizzando B set di training bootstrap e fare la media delle previsioni risultanti. Questi alberi crescono in profondità e non sono sottoposti al pruning, hanno un bias basso ed una varianza elevata ridotta dalla media calcolata

In generale, è stato dimostrato che il bagging offre miglioramenti impressionanti in termini di precisione combinando insieme centinaia o addirittura migliaia di alberi in un'unica procedura. Una delle implementazioni del bagging più utilizzate è la Random Forest.

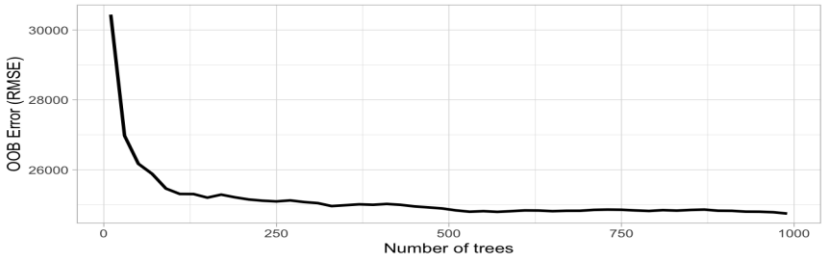
Random Forest

Le random forest vengono sviluppate utilizzando gli stessi principi fondamentali degli alberi decisionali e del bagging. Il bagging introduce una componente casuale nel processo di costruzione degli alberi sviluppandoli su copie bootstrap dei dati di training, aggregando le previsioni di tutti gli alberi riducendo la varianza dell'intera procedura, il che si traduce in migliori prestazioni predittive. Tuttavia, il bagging semplice produce una correlazione tra gli alberi che limita l'effetto della riduzione della varianza. Le random forest aiutano a ridurre la correlazione tra gli alberi conferendo più casualità al processo di crescita eseguendo la *split-variable randomization* in cui ogni volta che deve essere eseguita una divisione, la ricerca della split-variable è limitata a un sottoinsieme casuale di m_{try} delle p features originali. Il valore tipico per la regressione è $m_{try} = p/3$ ma può essere considerato un parametro di tuning. L'algoritmo di base per una random forest può essere generalizzato come segue:

1. Dato un dataset di training
2. Scelta del numero di alberi da sviluppare (n_trees)
3. for i=1 to n_trees do
4. | Generazione di un sample bootstrap dei dati originali
5. | Crescita dell'albero di regressione per i dati bootstrap
6. | for each split do
7. | | Selezione delle variabili m_{try} randomicamente dalle variabili p
8. | | Scelta della miglior variabile/split-point tra le m_{try}
9. | | Divisione del nodo in due figli
10. | end
11. | Utilizzo di un criterio di stop per determinare la completezza di un albero
12. end
13. Output degli alberi

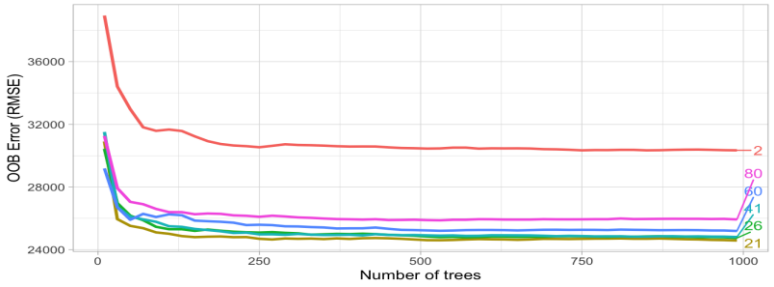
Iperparametri
Sebbene le foreste casuali funzionino bene di default, esistono diversi iperparametri da considerare durante l'addestramento di un modello. I principali iperparametri sono:

1. **Numero di alberi**: Sebbene tecnicamente non costituisca un iperparametro, il numero di alberi deve essere sufficientemente grande per stabilizzare il tasso di errore. Una *best practice* è utilizzare 10 volte il numero di features;

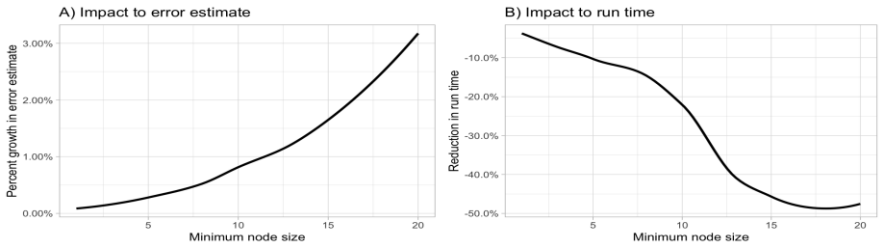


2. $\boldsymbol{m_{try}}$**:** controlla la feature *split-variable randomization* delle random forest e aiuta a bilanciare la correlazione dell'albero con una ragionevole forza predittiva. Con problemi di regressione il valore predefinito è $m_{try} = p/3$. Tuttavia, quando ci sono meno predittori rilevanti (dati rumorosi) un valore più alto di m_{try} tende a

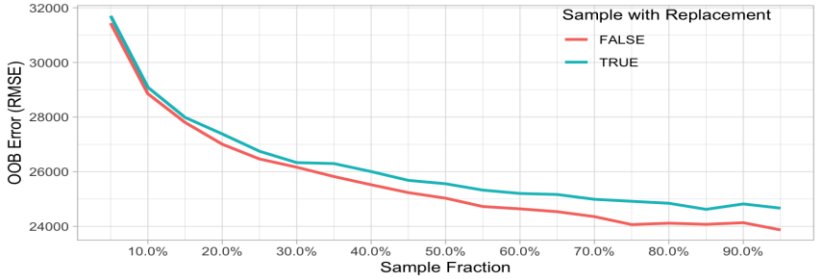
funzionare meglio perché rende più probabile la selezione delle features con il segnale più forte. Al contrario, quando sono presenti molti predittori rilevanti, un valore inferiore m_{try} potrebbe avere performance migliori.



3. **Complessità dell'albero**: Le random forest sono sviluppate su alberi decisionali individuali, di conseguenza la maggior parte delle implementazioni hanno uno o più iperparametri che consentono di controllare la profondità e la complessità dei singoli alberi, la dimensione del nodo, la profondità massima, il numero massimo di nodi terminali o la dimensione del nodo richiesta per consentire divisioni aggiuntive. La dimensione del nodo costituisce l'iperparametro più comune per controllare la complessità dell'albero e la maggior parte delle implementazioni utilizzano i valori predefiniti di 1 per la classificazione e 5 per la regressione. Tuttavia, se i dati possiedono molti predittori rumorosi e elevati valori di m_{try} hanno prestazioni migliori.



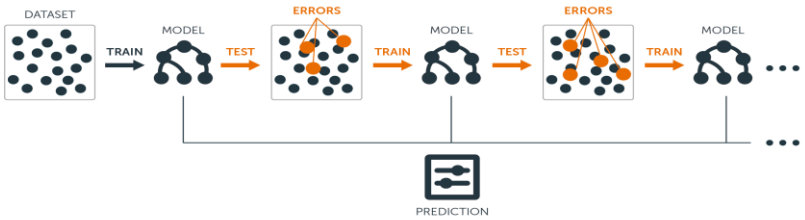
4. **Schema di sampling**: Lo schema predefinito è il bootstrap in cui il 100% delle osservazioni viene campionato con sostituzione. Tuttavia, è possibile modificare sia la dimensione del campione sia scegliere se campionare con o senza sostituzione. Il parametro della dimensione del sample determina quante osservazioni vengono considerate per il training di ogni albero. Diminuendo la dimensione del campione si ottengono alberi più diversificati e quindi una correlazione tra alberi inferiore, che può avere un effetto positivo sull'accuratezza della previsione. Di conseguenza, se ci sono alcune feature dominanti all'interno del dataset, la riduzione della dimensione del sample può aiutare a minimizzare la correlazione tra alberi.



5. **Regola di split**: la regola di split predefinita durante lo sviluppo degli alberi di una random forest consiste nel selezionare, tra tutti gli split-variable candidate, quella che minimizza la *Gini impurity* (classificazione) e l'SSE (regressione). Per aumentare l'efficienza computazionale, le regole di divisione possono essere randomizzate considerando solo un sottoinsieme casuale di possibili valori di splitting, questa procedura viene chiamata *extremely randomized tree*. A causa della maggiore casualità dei punti di splitting, questo metodo tende a non avere alcun miglioramento, o anche un impatto negativo, sull'accuratezza predittiva.

3.3.2 Boosting

L'idea principale del boosting è quella aggiungere nuovi modelli **in sequenza**, gestendo il compromesso bias-varianza partendo da un modello debole e aumenta sequenzialmente le sue prestazioni continuando a costruire nuovi alberi, dove ognuno nella sequenza cerca di rimediare agli errori commessi dal precedente focalizzandosi sulle istanze di training per cui l'albero precedente aveva maturato errori di previsione.



Componenti principali:

- **Base Learners**: il boosting è un framework che migliora iterativamente qualsiasi modello di apprendimento debole. Molte applicazioni di gradient boosting consentono di "collegare" tra loro varie classi di *weak learner*;
- **Training di weak models**: un *weak model* possiede un tasso di errore solo leggermente migliore rispetto all'ipotesi casuale. L'idea alla base del boosting risiede nel fatto che ogni modello nella sequenza migliora leggermente le prestazioni del precedente. Per quanto riguarda gli alberi decisionali, gli *shallow trees* (alberi con poche divisioni) rappresentano uno weak learner;
- **Training sequenziale rispetto agli errori**: gli alberi boosted vengono sviluppati in sequenza, utilizzando le informazioni di alberi precedenti per migliorare le prestazioni. Effettuando il fitting di ogni albero nella sequenza ai residui dell'albero precedente, si permette ai nuovi alberi della sequenza di concentrarsi sugli errori dell'albero precedente:

1. Fit di un albero di decisione ai dati: $F_1(x) = y$;
2. Fit del successivo albero decisionale ai residui del precedente: $h_1(X) = y - F_1(x)$;
3. Aggiunta del nuovo albero all'algoritmo: $F_2(x) = F_1(x) + h_1(x)$;
4. Fit dell'albero decisionale successivo ai residui di F_2: $h_2(x) = y - F_2(x)$;
5. Aggiunta del nuovo albero all'algoritmo: $F_3(x) = F_2(x) + h_2(x)$;
6. Ripere il processo fino a che un definito meccanismo (come la cross-validation) non determina il terminazione.

Il modello finale è un modello additivo stagewise di b alberi:

$$f(x) = \sum_{b=1}^{B} f^b(x)$$

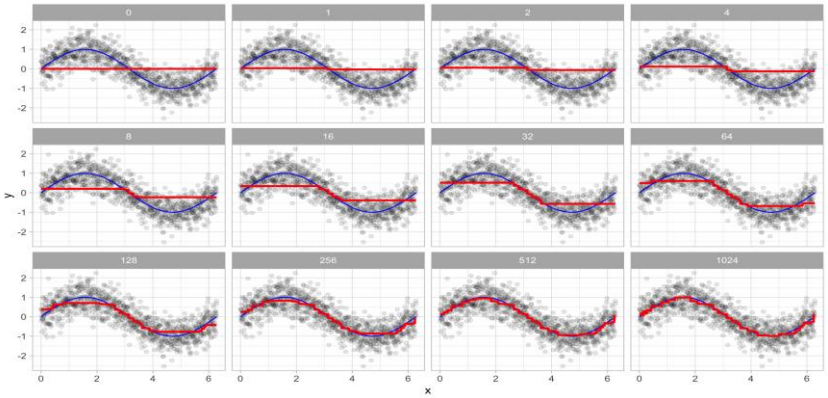


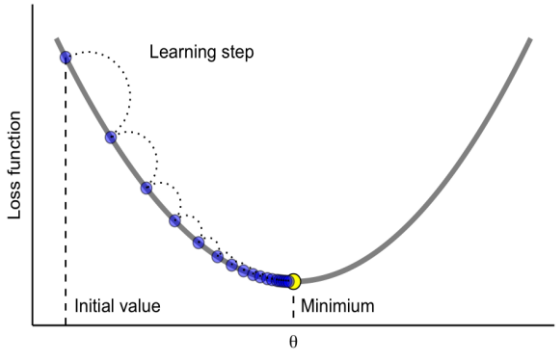
Illustrazione di come un singolo predittore (x) ha una relazione di forma sinusoidale (linea blu) con y insieme ad errori irriducibili. Il primo albero della serie è un singolo ceppo decisionale (albero con una singola divisione),

mentre ogni ceppo decisionale successivo è adattato ai residui del precedente. Inizialmente si riscontrano errori di grandi dimensioni, ma ogni albero decisionale aggiuntivo nella sequenza apporta un piccolo miglioramento in diverse aree dello spazio delle features dove gli errori sono ancora presenti.

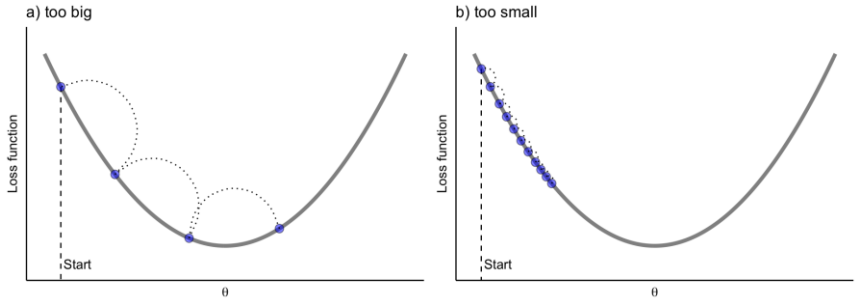
Gradient descent

Molti algoritmi di regressione, inclusi gli alberi decisionali, si concentrano sulla minimizzazione dei residui ed anche sulla funzione di perdita MSE. Questo approccio sfrutta il *gradient boosting* per minimizzare la funzione *mean squared error(MSE)*.

Gradient boosting è un algoritmo **gradient descent**, il quale rappresenta un metodo di ottimizzazione molto generico in grado di trovare soluzioni ottimali ad un'ampia gamma di problemi. L'idea generale della gradient descent è di modificare i parametri in modo iterativo per minimizzare la funzione di costo. In pratica ciò che fa questo algoritmo è misurare il gradiente locale della funzione di perdita (costo) per un dato insieme di parametri (θ) ed effettuare degli step nella direzione del gradient descent. Una volta che il gradiente è zero, si è raggiunto il minimo.

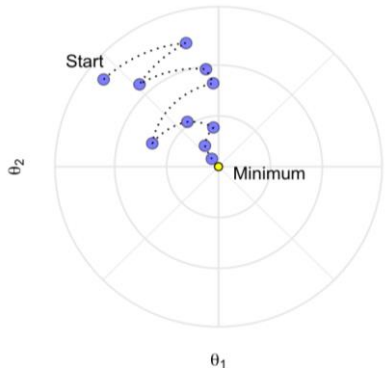


La gradient descent può essere eseguita su qualsiasi funzione di perdita differenziabile, ciò consente alle GBM di ottimizzare tale funzione. Un parametro importante nella gradient descent è la dimensione degli step controllata dal *learning rate*, se questo è troppo piccolo, l'algoritmo richiederà molte iterazioni per trovare il minimo, al contrario, lo si potrebbe oltrepassare.



Con un learning rate basso saranno necessarie molte iterazioni per trovare il minimo, mentre con un learning rate alto c'è il rischio di oltrepassarlo.

Inoltre, non tutte le funzioni di costo sono graficamente *convesse*. Potrebbero esserci minimi locali e altre sezioni irregolari i quali rendono difficile trovare il minimo globale. Lo *Stochastic gradient descent* può essere utile ad affrontare questo tipo di problemi campionando una frazione delle osservazioni di training (senza sostituzione) e sviluppando l'albero successivo utilizzando quel subsample. Questo processo rende l'algoritmo più veloce, ma la natura stocastica del campionamento randomico aggiunge anche una certa casualità nella discesa del gradiente della funzione di perdita. Sebbene questa casualità non consenta all'algoritmo di trovare il minimo globale assoluto, può aiutare ad evitare i minimi locali per avvicinarsi sufficientemente al minimo globale.



Il GBM è una tecnica piuttosto flessibile ed esistono diverse possibilità di configurazione per gli iperparametri:

Iperparametri
Un semplice modello GBM contiene due categorie di iperparametri: *boosting hyperparameter* e *tree-specific hyperparameter*.
I due principali **boosting hyperparameter** sono:

- **Numero di alberi**: il numero totale di alberi nella sequenza. La media degli alberi cresciuti in modo indipendente nel bagging o random forest rende molto difficile incorrere in overfitting con molti alberi. Tuttavia, i GBM funzionano in modo diverso poiché ogni albero viene sviluppato in sequenza per correggere gli errori dell'albero precedente. Ad esempio, nella regressione, i GBM si focalizzano sui residui finchè ne avranno la possibilità. Inoltre, a seconda dei valori degli altri iperparametri, i GBM richiedono spesso molti alberi (è possibile anche avere migliaia di alberi) ma poiché possono facilmente andare in overfitting è necessario trovare il numero ottimale di alberi che minimizzi la funzione di perdita tramite cross-validation;
- **Learning rate**: determina il contributo di ogni albero sul risultato finale e controlla la velocità con cui l'algoritmo procede lungo il gradient descent (apprendimento). I valori vanno da 0–1 con valori tipici compresi tra 0,001–0,3. Valori più piccoli rendono il modello robusto alle caratteristiche specifiche di ogni singolo albero, permettendogli una buona generalizzazione e facilitando l'arresto prima dell'overfitting. Tuttavia questi valori aumentano il rischio di non raggiungere l'optimum con un numero costante di alberi e sono più computazionalmente onerosi. Questo iperparametro è anche chiamato *shrinkage*. In generale, più piccolo è questo valore, più accurato può essere il modello, ma richiederà più alberi nella sequenza.

I due principali **tree-specific hyperparameter** sono:

- **Profondità dell'albero**: gestisce la profondità dei singoli alberi. I valori tipici hanno una profondità di 3–8 ma non è raro riscontrare una profondità pari a 1. Alberi meno profondi come i ceppi decisionali sono computazionalmente efficienti (ma richiedono un maggior numero di alberi). Tuttavia, alberi più profondi consentono all'algoritmo di acquisire interazioni uniche ma aumentano anche il rischio di overfitting;
- **Numero minimo di osservazioni nei nodi terminali**: controlla la complessità di ogni albero. Tipicamente oscilla tra 5 e 15, dove valori più alti aiutano a prevenire che un modello possa apprendere relazioni altamente specifiche per il particolare sample selezionato per un albero (overfitting) ma valori più piccoli possono aiutare con classi target sbilanciate nei problemi di classificazione.

4.CASO DI STUDIO

Nel presente caso di studio verrà considerato il dataset AmesHousing reperito nel repository del CRAN (https://CRAN.R-project.org/package=AmesHousing). Per implementare gli alberi di regressione ed i relativi algoritmi di stima relativi alle tecniche di bagging e boosting sono state utilizzate le seguenti librerie:
-Rpart;
-Caret;
-iPred;
-RandomForest;
-Ranger;

-H2o;
-Gbm;
-XgBoost;

```
packages <-
c("AmesHousing","rsample","rpart","rpart.plot","dplyr","caret","ipred","randomForest","dplyr","ggplo
t2","ranger","h2o","gbm","xgboost","pdp","lime","vtreat")
not_installed <- packages[!(packages %in% installed.packages()[ , "Package"])]
if(length(not_installed)) install.packages(not_installed)

library(AmesHousing)
library(rsample)
library(rpart)
library(rpart.plot)
library(dplyr)
library(caret)
library(ipred)
library(randomForest)
library(dplyr)
library(ggplot2)
library(ranger)
library(h2o)
library(gbm)
library(xgboost)
library(pdp)
library(lime)
library(vtreat)
```

DATA SPLITTING

Prima di effettuare qualsiasi tipo di operazione è necessario procedere con lo splitting del dataset.

```
# Creazione dei set di training e test (70-30) utilizzando set.seed per riproducibilità
set.seed(123)
ames_split <- initial_split(AmesHousing::make_ames(), prop = .7)
ames_train <- training(ames_split)
ames_test  <- testing(ames_split)
```

REGRESSION TREE IMPLEMENTATION

Esistono diversi metodi per implementare gli alberi di regressione, ma uno dei più noti è il *CART* (classification and regression tree). Per effettuare il fit di un albero di regressione può essere utilizzata la libreria *rpart* e *rpart.plot* per la sua visualizzazione. La libreria **Rpart** (Recursive Partitioning And Regression Trees) rappresenta l'implementazione naturale dell'algoritmo CART, effettuando splitting ricorsivi del dataset fino al soddisfacimento di un criterio di stop. Ogni split viene effettuato sulla base di una variabile indipendente, con l'obiettivo di minimizzare l'eterogeneità della variabile dipendente.

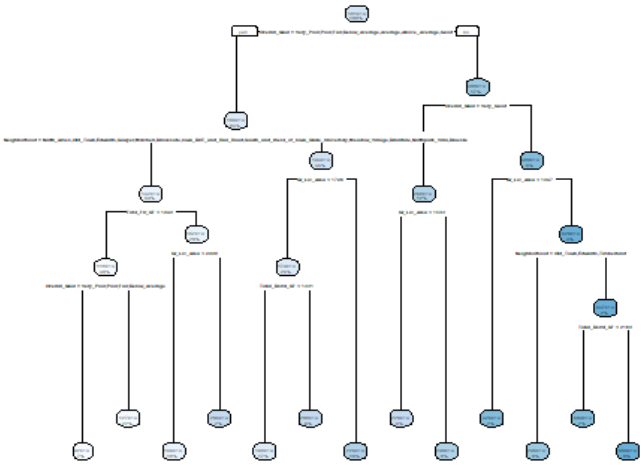
```
t <- rpart(
  formula = Sale_Price ~ .,
  data    = ames_train,
  method  = "anova"  #necessario per alberi di regressione
  )
t

## n= 2051
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 2051 1.273987e+13 180775.50
##    2) Overall_Qual=Very_Poor,Poor,Fair,Below_Average,Average,Above_Average,Good 1703 4.032269e+12
156431.40
```

```
##      4)
Neighborhood=North_Ames,Old_Town,Edwards,Sawyer,Mitchell,Brookside,Iowa_DOT_and_Rail_Road,South_and_
West_of_Iowa_State_University,Meadow_Village,Briardale,Northpark_Villa,Blueste 1015 1.360332e+12
131803.50
##        8) First_Flr_SF< 1048.5 611 4.924281e+11 118301.50
##         16) Overall_Qual=Very_Poor,Poor,Fair,Below_Average 152 1.053743e+11  91652.57 *
##         17) Overall_Qual=Average,Above_Average,Good 459 2.433622e+11 127126.40 *
##        9) First_Flr_SF>=1048.5 404 5.880574e+11 152223.50
##         18) Gr_Liv_Area< 2007.5 359 2.957141e+11 145749.50 *
##         19) Gr_Liv_Area>=2007.5 45 1.572566e+11 203871.90 *
##      5)
Neighborhood=College_Creek,Somerset,Northridge_Heights,Gilbert,Northwest_Ames,Sawyer_West,Crawford,T
imberland,Northridge,Stone_Brook,Clear_Creek,Bloomington_Heights,Veenker,Green_Hills 688
1.148069e+12 192764.70
##       10) Gr_Liv_Area< 1725.5 482 5.162415e+11 178531.00
##         20) Total_Bsmt_SF< 1331 352 2.315412e+11 167759.00 *
##         21) Total_Bsmt_SF>=1331 130 1.332603e+11 207698.30 *
##       11) Gr_Liv_Area>=1725.5 206 3.056877e+11 226068.80 *
##    3) Overall_Qual=Very_Good,Excellent,Very_Excellent 348 2.759339e+12 299907.90
##      6) Overall_Qual=Very_Good 249 9.159879e+11 268089.10
##       12) Gr_Liv_Area< 1592.5 78 1.339905e+11 220448.90 *
##       13) Gr_Liv_Area>=1592.5 171 5.242201e+11 289819.70 *
##      7) Overall_Qual=Excellent,Very_Excellent 99 9.571896e+11 379937.20
##       14) Gr_Liv_Area< 1947 42 7.265064e+10 325865.10 *
##       15) Gr_Liv_Area>=1947 57 6.712559e+11 419779.80
##         30) Neighborhood=Old_Town,Edwards,Timberland 7 8.073100e+10 295300.00 *
##         31) Neighborhood=College_Creek,Somerset,Northridge_Heights,Northridge,Stone_Brook 50
4.668730e+11 437207.00
##           62) Total_Bsmt_SF< 2168.5 40 1.923959e+11 408996.90 *
##           63) Total_Bsmt_SF>=2168.5 10 1.153154e+11 550047.30 *
```

Analizzando gli step degli split è possibile notare che nel nodo radice sono presenti 2051 osservazioni e la prima variabile su cui si effettua la suddivisione è *Overall_Qual*. Al primo nodo tutte le osservazioni *Overall_Qual=Very_Poor,Poor,Fair,Below_Average,Average,Above_Average,Good* vanno al secondo ramo. Vengono inoltre mostrati il numero totale di osservazioni del ramo (1703), il loro prezzo di vendita medio (156431.40) e SSE (4.032269e+12). Da queste informazioni si deduce che la variabile più importante che raggiunge la maggiore riduzione del SSE è Overall_Qual con le abitazioni all'estremità superiore che hanno quasi il doppio del prezzo medio di vendita.

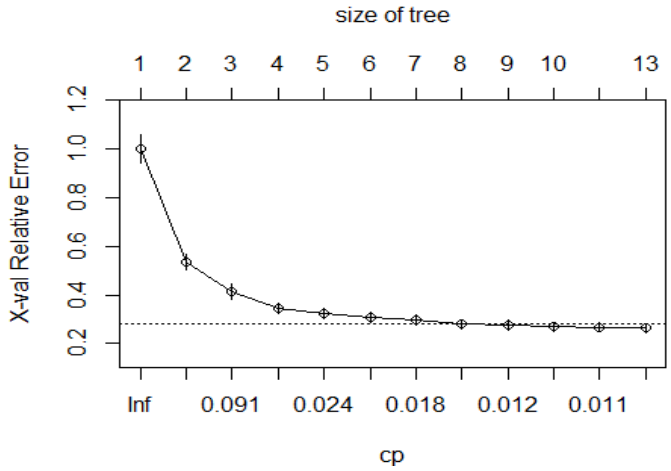
Tramite *rpart.plot* è possibile visualizzare il modello tramite un grafico nel quale vengono evidenziate la percentuale di dati che ricade in ogni nodo e il prezzo medio di vendita per quel ramo. Questo albero contiene 11 nodi interni che portano a 12 nodi terminali.

```
rpart.plot(t)
```



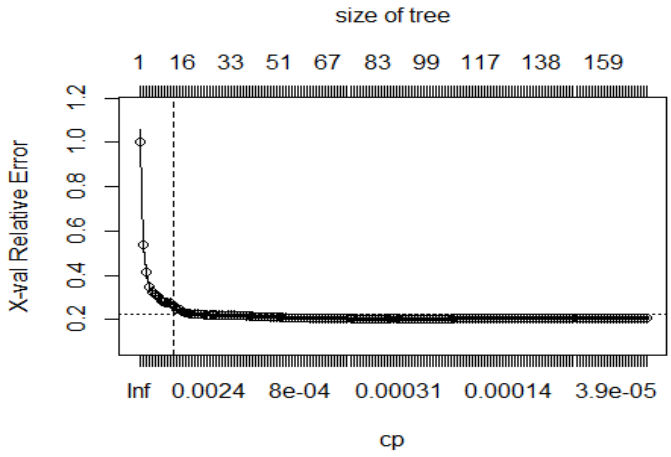
Rpart applica automaticamente una serie di valori di complessità dei costi α per potare l'albero. Per confrontare l'errore di ciascun valore α, rpart esegue una 10-fold cross-validation in modo che l'errore associato a un dato α venga calcolato sui dati di validazione. In questo caso i risultati decrescono dopo 12 nodi terminali (dove y è l'errore di cross-validation, l'asse x inferiore rappresenta la complessità dei costi α e l'asse x superiore è il numero di nodi terminali (dimensione albero = |T|). La linea tratteggiata indica che utilizzando un albero con 8 nodi terminali ci si potrebbero aspettare risultati simili a quelli ottenuti in precedenza.

```
plotcp(t)
```



Per motivare la scelta di 12 nodi terminali è possibile forzare rpart a generare un albero completo utilizzando *cp=0*. Si può notare che dopo 12 nodi terminali si riscontrano rendimenti decrescenti nella riduzione degli errori man mano che l'albero cresce più in profondità.

```
t2 <- rpart(
    formula = Sale_Price ~ .,
    data    = ames_train,
    method  = "anova",
    control = list(cp = 0, xval = 10)
)

plotcp(t2)
abline(v = 12, lty = "dashed")
```



Di default rpart esegue un tuning automatico, definendo una sottostruttura ottimale di 12 split, 12 nodi terminali e un errore di cross-validation di 0,263. Tuttavia, è possibile eseguire ulteriori ottimizzazioni per provare a migliorare le prestazioni del modello.

```
t$cptable
```

```
##            CP nsplit rel error    xerror       xstd
## 1  0.46690132      0 1.0000000 1.0009222 0.05855161
## 2  0.11961409      1 0.5330987 0.5347929 0.03116217
## 3  0.06955813      2 0.4134846 0.4151417 0.03058554
## 4  0.02559992      3 0.3439265 0.3461258 0.02207839
## 5  0.02196620      4 0.3183265 0.3242197 0.02182111
## 6  0.02023390      5 0.2963603 0.3074877 0.02129292
## 7  0.01674138      6 0.2761264 0.2963372 0.02106996
## 8  0.01188709      7 0.2593850 0.2795199 0.01903482
## 9  0.01127889      8 0.2474980 0.2762666 0.01936472
## 10 0.01109955      9 0.2362191 0.2699895 0.01902217
## 11 0.01060346     11 0.2140200 0.2672133 0.01883219
## 12 0.01000000     12 0.2034165 0.2635207 0.01881691
```

Tuning

Oltre al parametro di complessità dei costi (α) è necessario anche ottimizzare:

- **minsplit**: il numero minimo di dati necessari per effettuare uno split prima di dover creare un nodo terminale.
- **maxdepth**: il numero massimo di nodi interni tra il nodo radice e i nodi terminali.

Rpart utilizza l'argomento *control* per definire un elenco di valori da settare per gli iperparametri, per non valutare più modelli manualmente è possibile eseguire una *grid-search* per effettuare una ricerca automatica ed identificare la configurazione ottimale. Per eseguire una grid-search serve creare una griglia iperparametrica:

```
hyper_grid <- expand.grid(
  minsplit = seq(5, 10, 1),
  maxdepth = seq(8, 15, 1) #da 8 a 15 dato che la profondità del modello originale era 12
)

head(hyper_grid)
```

```
##   minsplit maxdepth
## 1        5        8
## 2        6        8
## 3        7        8
## 4        8        8
## 5        9        8
## 6       10        8
```

```
#numero totale di combinazioni
nrow(hyper_grid)
```

```
## [1] 48
```

Per automatizzare questo processo si può definire un ciclo for iterando attraverso ciascuna combinazione minsplit e maxdepth.

```
models <- list()

for (i in 1:nrow(hyper_grid)) {

  # ottenere i valori di minsplit, maxdepth alla i-esima riga
  minsplit <- hyper_grid$minsplit[i]
  maxdepth <- hyper_grid$maxdepth[i]

  # train del modello e salvataggio nella lista
  models[[i]] <- rpart(
    formula = Sale_Price ~ .,
```

```
    data    = ames_train,
    method  = "anova",
    control = list(minsplit = minsplit, maxdepth = maxdepth)
    )
}
```

Si crea dunque funzione per estrarre l'errore minimo associato al valore ottimale della complessità dei costi (α) per ogni modello. Filtrando per i primi 5 valori di errore minimo, è possibile notare che il modello ottimale ottiene prestazioni paragonabili al modello precedente.

```
# funzione per ottenere cp ottimale
get_cp <- function(x) {
  min    <- which.min(x$cptable[, "xerror"])
  cp <- x$cptable[min, "CP"]
}

# funzione per ottenere l'errore minimo
get_min_error <- function(x) {
  min    <- which.min(x$cptable[, "xerror"])
  xerror <- x$cptable[min, "xerror"]
}

hyper_grid %>%
  mutate(
    cp    = purrr::map_dbl(models, get_cp),
    error = purrr::map_dbl(models, get_min_error)
    ) %>%
  arrange(error) %>%
  top_n(-5, wt = error)

##   minsplit maxdepth         cp     error
## 1        7       14 0.01000000 0.2655860
## 2        8       14 0.01000000 0.2660555
## 3        8       15 0.01000000 0.2674049
## 4        8       12 0.01000000 0.2677339
## 5        7       13 0.01060346 0.2682152
```

L'RMSE finale è 39852.01, il che suggerisce che, in media, i prezzi di vendita previsti sono di circa $ 39,852 in meno rispetto al prezzo di vendita effettivo.

```
optimal_tree <- rpart(
    formula = Sale_Price ~ .,
    data    = ames_train,
    method  = "anova",
    control = list(minsplit = 11, maxdepth = 8, cp = 0.01)
    )

pred <- predict(optimal_tree, newdata = ames_test)
RMSE(pred = pred, obs = ames_test$Sale_Price)

## [1] 39852.01
```

SIMPLE BAGGING

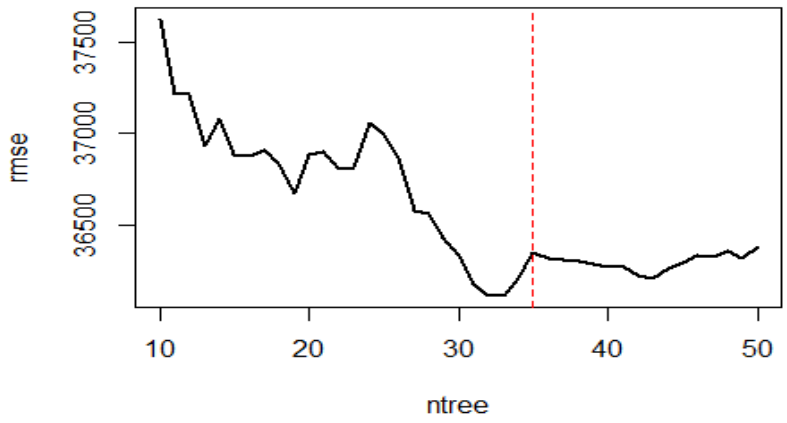
ipred

Ipred (Improved predictors) è un package utilizzato per il bagging in contesti sia di classificazione che di regressione e rappresenta la prima interfaccia unificata per i predittori e stimatori di errori. Per effettuare il fitting di un modello, invece di usare rpart si può *ipred::bagging*, con *coob = TRUE* per stimare l'errore di test. Si può notare che l'errore di stima iniziale è circa $3K in meno rispetto all'errore di test che ottenuto con un singolo albero ottimale (36991 contro 39852).

```r
# rendere il bootstrap riproducibile
set.seed(123)

# train del modello bagged
bagged_t <- bagging(
  formula = Sale_Price ~ .,
  data    = ames_train,
  coob    = TRUE
)

bagged_t

##
## Bagging regression trees with 25 bootstrap replications
##
## Call: bagging.data.frame(formula = Sale_Price ~ ., data = ames_train,
##     coob = TRUE)
##
## Out-of-bag estimate of root mean squared error:  36991.67
```

Valutando il modello con un numero crescente di alberi si può osservare che all'inizio è presente una drastica riduzione della varianza (e dell'errore) seguita da una stabilizzazione dei valori.

```r
# valutazione 10-50 alberi bagged
ntree <- 10:50

# creazione di un vettore vuoto per memorizzare i valori OOB dell'RMSE
rmse <- vector(mode = "numeric", length = length(ntree))

for (i in seq_along(ntree)) {
  # riproducibilità
  set.seed(123)

  # modello bagged
  model <- bagging(
  formula = Sale_Price ~ .,
  data    = ames_train,
  coob    = TRUE,
  nbagg   = ntree[i]
)
  # ottenere errore OOB
  rmse[i] <- model$err
}

plot(ntree, rmse, type = 'l', lwd = 2)
abline(v = 35, col = "red", lty = "dashed")
```



caret Il pacchetto caret (Classification And REgression Training) contiene funzioni per semplificare il processo di training del modello per problemi di regressione e classificazione complessi. Sebbene sia possibile utilizzare l'errore OOB, l'esecuzione della cross-validation fornisce anche una migliore comprensione del vero errore di test previsto. Inoltre è possibile valutare l'importanza delle variabili per gli alberi. Caret possiede diverse funzioni che tentano di semplificare il processo di costruzione e valutazione del modello, così come la selezione delle feature e altre tecniche.

Uno degli strumenti principali nel pacchetto è la funzione *train* che può essere utilizzata per:

- Valutare, utilizzando il ricampionamento, l'effetto dei parametri di tuning del modello sulle prestazioni
- Scegliere il modello ottimale tra questi parametri
- Stimare le prestazioni del modello per un set di training

Più formalmente:

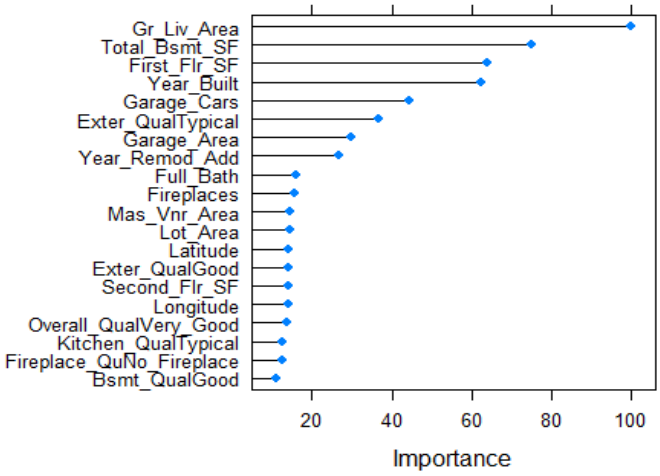
Definizione di insiemi di valori di parametri del modello da valutare
for each parametro settato do
| for each iterazione di ricampionamento do
| | Mantenere determinati samples
| | Pre-process dei dati (opzionale)
| | Fit del modello
| | Predizione dei samples mantenuti
| end
| Calcolo della performance media tra le predizioni ottenute
end
Determinare il parametro ottimale
Fit del modello finale su tutti i dati di training

In questo caso si esegue un modello usando una 10-fold cross-validation, osservando che l'RMSE è $ 35.854
L'importanza di una variabile per gli alberi di regressione viene misurata valutando la quantità totale di SSE persa tramite gli split su un dato predittore e mediata su tutti gli m alberi. I predittori con il maggiore impatto medio sul SSE sono considerati i più importanti.

```
# 10-fold cross validation
ctrl <- trainControl(method = "cv",  number = 10)

# CV modello bagged
bagged_cv <- train(
  Sale_Price ~ .,
  data = ames_train,
  method = "treebag",
  trControl = ctrl,
  importance = TRUE
  )

# valutazione risultati
bagged_cv
```

```
## Bagged CART
##
## 2051 samples
##   80 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 1846, 1845, 1846, 1845, 1847, 1847, ...
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   35854.02  0.8009063  23785.85
```

```r
# plot delle variabili più importanti
plot(varImp(bagged_cv), 20)
```



Confrontandolo con il test impostato sul sample, si potrà notare che la stima dell'errore con cross-validation è molto vicina. Si è ridotto con successo l'errore a circa $ 35.000.

```r
pred <- predict(bagged_cv, ames_test)
RMSE(pred, ames_test$Sale_Price)

## [1] 35357.89
```

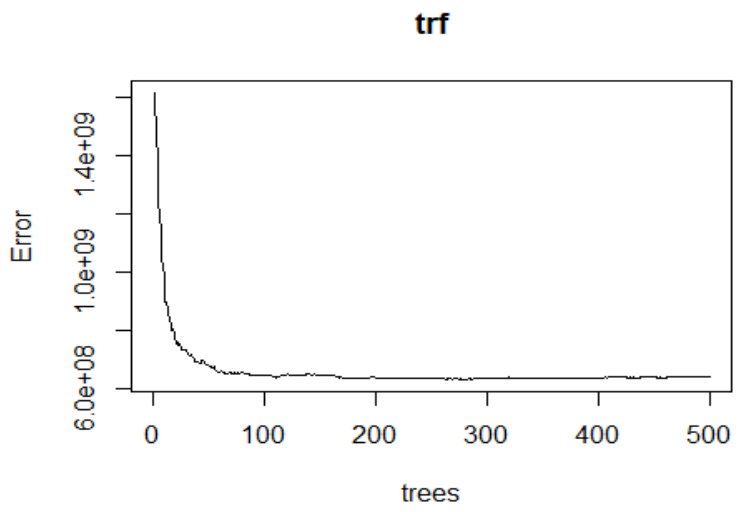
RANDOM FOREST

Per implementare questo algoritmo di bagging si utilizzeranno le librerie *randomForest,ranger* e *h20*.
Il pacchetto **randomForest** è il più conosciuto ed utilizzato per quanto riguarda l'implementazione di quest'algoritmo, nonostante non fornisca performance elevate con dataset di grandi dimensioni. La random forest standard esegue 500 alberi e $features/3 = 26$ variabili predittive selezionate casualmente ad ogni split. La media su tutti i 500 alberi fornisce un OOB MSE=639516350 (RMSE=25288).

```r
# per riproducibilità
set.seed(123)

# Random forest standard
trf <- randomForest(
  formula = Sale_Price ~ .,
  data    = ames_train
)

trf

##
## Call:
##  randomForest(formula = Sale_Price ~ ., data = ames_train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 26
##
##           Mean of squared residuals: 639516350
##                     % Var explained: 89.7
```

Si può definire graficamente il modello evidenziando il tasso di errore calcolandone la media su più alberi. Il tasso di errore si stabilizza utilizzando circa 100 alberi.

```r
plot(trf)
```



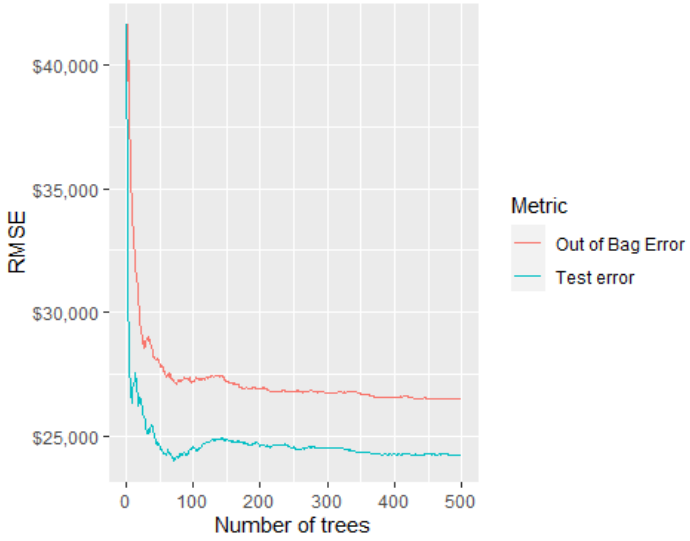
Nel grafico si evidenzia l'errore OOB il quale consente di trovare il numero di alberi ottimale per ottenere il tasso di errore più basso (280) e un errore medio del prezzo di vendita di $ 25.135.

```
#numero di alberi con MSE più basso
which.min(trf$mse)

## [1] 280

#RMSE del random forest ottimale
sqrt(trf$mse[which.min(trf$mse)])

## [1] 25135.88
```

randomForest consente inoltre di utilizzare un validation set per misurare l'accuratezza predittiva. In questo caso il training set è stato ulteriormente suddiviso per ottenere un set di training e validation.

```
# creazione set di training e validation
set.seed(123)
valid_split <- initial_split(ames_train, .8)

# training
ames_train_v2 <- analysis(valid_split)

# validation
ames_valid <- assessment(valid_split)
x_test <- ames_valid[setdiff(names(ames_valid), "Sale_Price")]
y_test <- ames_valid$Sale_Price

rf_oob_comp <- randomForest(
  formula = Sale_Price ~ .,
  data    = ames_train_v2,
  xtest   = x_test,
  ytest   = y_test
)

# estrazione OOB & errori di validation
oob <- sqrt(rf_oob_comp$mse)
validation <- sqrt(rf_oob_comp$test$mse)

# comparazione errori
tibble::tibble(
  `Out of Bag Error` = oob,
  `Test error` = validation,
```

```
  ntrees = 1:rf_oob_comp$ntree
) %>%
  gather(Metric, RMSE, -ntrees) %>%
  ggplot(aes(ntrees, RMSE, color = Metric)) +
  geom_line() +
  scale_y_continuous(labels = scales::dollar) +
  xlab("Number of trees")
```



Le random forest sono uno dei migliori algoritmi di apprendimento automatico "out of the box" e per questo in genere richiedono poche operazioni di tuning. Tuttavia, è possibile cercare di migliorare il modello.

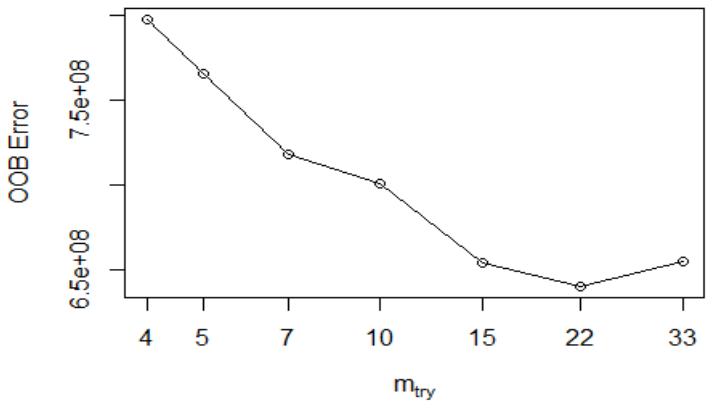
Tuning

Gli iperparametri da considerare nella fase di ottimizzazione sono:

- **nrtee**: Sono necessari alberi sufficienti a stabilizzare l'errore ma l'uso di troppi alberi può risultare inutile ed inefficiente, specialmente con grandi dataset;
- **mtry**: il numero di variabili da campionare casualmente come candidati per ogni split. Quando mtry=p il modello è equivalente al bagging, mentre quando mtry=1 la variabile di split è casuale, quindi tutte le variabili hanno la stessa probabilità ma ciò può portare a risultati distorti;
- **samplesize**: il numero di campioni su cui effettuare training. Il valore predefinito è il 63,25% del training set poiché questo è il valore atteso di osservazioni univoche nel campione bootstrap. Dimensioni inferiori del campione riducono il tempo di training ma possono introdurre bias, mentre l'aumento della dimensione del campione può incrementare le prestazioni ma a rischio di overfitting introducendo più varianza. Tipicamente viene configurato nell'intervallo 60-80%;
- **nodesize**: rappresenta il numero minimo di campioni nei nodi terminali e controlla la complessità degli alberi. Una dimensione del nodo più piccola consente di creare alberi più profondi e complessi ma che introducono più varianza causando overfitting, mentre alberi meno profondi introducono più bias;
- **maxnodes**: rappresenta il numero massimo di nodi terminali ed è un altro modo per controllare la complessità degli alberi. Più nodi equivalgono ad alberi più profondi e complessi e viceversa.
  Il parametro mtry viene ottimizzato usando *randomForest::tuneRF* il quale fornirà un valore che aumenterà per un determinato fattore di incremento fino a quando l'errore OOB smetterà di incrementare di un soglia specificata.

```
# nomi delle features
features <- setdiff(names(ames_train), "Sale_Price")

set.seed(123)

trf2 <- tuneRF(
```

```
  x          = ames_train[features],
  y          = ames_train$Sale_Price,
  ntreeTry   = 500,
  mtryStart  = 5,
  stepFactor = 1.5,
  improve    = 0.01,
  trace      = FALSE
)

## -0.04236505 0.01
## 0.0614441 0.01
## 0.02425961 0.01
## 0.06634214 0.01
## 0.02149491 0.01
## -0.02257957 0.01
```



E' possibile effettuare una grid search per testare tutte le combinazioni degli iperparametri e valutare il modello. In questo caso randomForest diventa piuttosto inefficiente perciò viene utilizzato **ranger**.
Questo pacchetto offre un'implementazione più rapida di random forest, particolarmente adatto per dati ad alta dimensionalità. Supporta inoltre classificazione, regressione e le survival forest. Vengono utilizzati due diversi algoritmi di split, il primo ordina i valori delle features in anticipo e vi accede in base al loro indice mentre nel secondo i valori grezzi vengono recuperati e ordinati durante la divisione. L'efficienza della memoria è ottenuta evitando la produzione di copie dei dati originali, salvando le informazioni dei nodi in strutture dati semplici e liberando la memoria in anticipo.

```
# velocità di randomForest
system.time(
  ames_randomForest <- randomForest(
    formula = Sale_Price ~ .,
    data    = ames_train,
    ntree   = 500,
    mtry    = floor(length(features) / 3)
  )
)

##    user  system elapsed
##   50.13    0.06   50.20

# velocità di ranger
system.time(
  ames_ranger <- ranger(
    formula   = Sale_Price ~ .,
    data      = ames_train,
    num.trees = 500,
    mtry      = floor(length(features) / 3)
```

```
  )
)

##    user  system elapsed
##    9.97    0.05    1.39
```

Per eseguire la grid search è necessario definire la griglia di iperparametri.

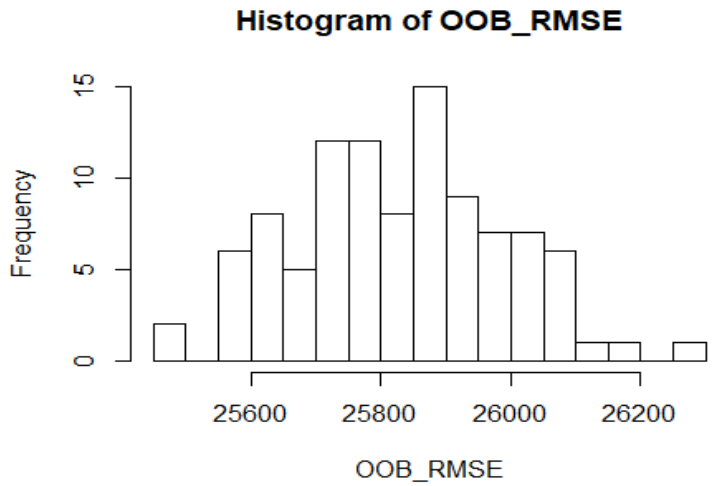
```
# grid search
hyper_grid <- expand.grid(
  mtry       = seq(20, 30, by = 2),
  node_size  = seq(3, 9, by = 2),
  sampe_size = c(.55, .632, .70, .80),
  OOB_RMSE   = 0
)

# numero totale di combinazioni
nrow(hyper_grid)

## [1] 96
```

Si cicla su ogni combinazione utilizzando 500 alberi. Il OOB RMSE varia tra ~ 25.000-26.000 ed i risultati mostrano che i modelli con dimensioni del campione leggermente più grandi (70-80%) e alberi più profondi ottengano prestazioni migliori.

```
for(i in 1:nrow(hyper_grid)) {

  # train
  model <- ranger(
    formula         = Sale_Price ~ .,
    data            = ames_train,
    num.trees       = 500,
    mtry            = hyper_grid$mtry[i],
    min.node.size   = hyper_grid$node_size[i],
    sample.fraction = hyper_grid$sampe_size[i],
    seed            = 123
  )

  # aggiunta errore OOB alla griglia
  hyper_grid$OOB_RMSE[i] <- sqrt(model$prediction.error)
}

hyper_grid %>%
  dplyr::arrange(OOB_RMSE) %>%
  head(10)

##    mtry node_size sampe_size OOB_RMSE
## 1    28         3        0.8 25477.32
## 2    28         5        0.8 25543.14
## 3    28         7        0.8 25689.05
## 4    28         9        0.8 25780.86
## 5    30         3        0.8 25818.27
## 6    24         3        0.8 25838.55
## 7    26         3        0.8 25839.71
## 8    20         3        0.8 25862.25
## 9    30         5        0.8 25884.35
## 10   24         5        0.8 25895.22
```

Dal momento che il miglior modello è stato ottenuto con mtry=28 e node_size=3 è possibile ripetere l'operazione per ottenere una migliore aspettativa del tasso di errore.

```r
OOB_RMSE <- vector(mode = "numeric", length = 100)

for(i in seq_along(OOB_RMSE)) {

  optimal_ranger <- ranger(
    formula         = Sale_Price ~ .,
    data            = ames_train,
    num.trees       = 500,
    mtry            = 28,
    min.node.size   = 3,
    sample.fraction = .8,
    importance      = 'impurity'
  )

  OOB_RMSE[i] <- sqrt(optimal_ranger$prediction.error)
}

hist(OOB_RMSE, breaks = 20)
```



Il pacchetto **h2o** è un'interfaccia basata su Java potente ed efficiente. Le sue caratteristiche principali sono:

- Calcolo distribuito e parallelizzato su un singolo nodo o un cluster multi-nodo;
- Arresto anticipato automatico basato sulla convergenza delle metriche con la tolleranza relativa specificate dall'utente;
- Supporto per famiglie esponenziali (Poisson, Gamma, Tweedie) e funzioni di perdita oltre alle distribuzioni binomiale (Bernoulli), Gaussiana e multinomiale;
- Grid search per l'ottimizzazione degli iperparametri e la selezione del modello;
- Data-distributed, il che significa che l'intero dataset non ha bisogno di adattarsi alla memoria su un singolo nodo, quindi si adatta a set di training di ogni dimensione;
- Utilizza l'errore quadratico per determinare gli split ottimali;
- Alberi multiclasse costruiti in parallelo;
- Licenza Apache 2.0;
  E' possibile definire una *complete grid search*, esaminando ogni combinazione specificata tramite *hyper_grid.h2o*. In questo caso si cerca tra 96 modelli ma poiché viene eseguita una ricerca cartesiana completa, questo processo non è più veloce di quello effettuato in precedenza. Tuttavia si ottengono prestazioni con un OOB RMSE di 24349 ($\sqrt{5.9292}$ E8), che è un risultato migliore rispetto a quelli ottenuti finora. Questo per via della configurazione predefinita dei parametri in h20.

```r
h2o.no_progress()
h2o.init(min_mem_size  = "12g")
```

```
##
## H2O is not running yet, starting it now...
##
## Note:  In case of errors look at the following log files:
##     C:\Users\polic\AppData\Local\Temp\RtmpuCLHKt\file358817c46d7/h2o_polic_started_from_r.out
##     C:\Users\polic\AppData\Local\Temp\RtmpuCLHKt\file35882aad6acf/h2o_polic_started_from_r.err
##
##
## Starting H2O JVM and connecting:  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         4 seconds 310 milliseconds
##     H2O cluster timezone:       Europe/Berlin
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.30.1.1
##     H2O cluster version age:    1 month and 1 day
##     H2O cluster name:           H2O_started_from_R_polic_doo177
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   11.31 GB
##     H2O cluster total cores:    8
##     H2O cluster allowed cores:  8
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         Amazon S3, Algos, AutoML, Core V3, TargetEncoder, Core V4
##     R Version:                  R version 3.6.1 (2019-07-05)

# creazione nomi delle features
y <- "Sale_Price"
x <- setdiff(names(ames_train), y)

# training set in oggetto h2o
train.h2o <- as.h2o(ames_train)

# griglia di iperparametri
hyper_grid.h2o <- list(
  ntrees      = seq(200, 500, by = 100),
  mtries      = seq(20, 30, by = 2),
  sample_rate = c(.55, .632, .70, .80)
)

# grid search
grid <- h2o.grid(
  algorithm = "randomForest",
  grid_id = "rf_grid",
  x = x,
  y = y,
  training_frame = train.h2o,
  hyper_params = hyper_grid.h2o,
  search_criteria = list(strategy = "Cartesian")
  )

# collezione dei risultati e ordinamento per metrica di performance
grid_perf <- h2o.getGrid(
  grid_id = "rf_grid",
  sort_by = "mse",
  decreasing = FALSE
  )
print(grid_perf)
```

```
## H2O Grid Details
## ================
##
## Grid ID: rf_grid
## Used hyper parameters:
##   -  mtries
##   -  ntrees
##   -  sample_rate
## Number of models: 96
## Number of failed models: 0
##
## Hyper-Parameter Search Summary: ordered by increasing mse
##   mtries ntrees sample_rate        model_ids                 mse
## 1     22    300         0.8 rf_grid_model_80 5.929264984447558E8
## 2     30    300         0.8 rf_grid_model_84 5.948423191807088E8
## 3     26    500         0.8 rf_grid_model_94 5.973273894450601E8
## 4     26    400         0.8 rf_grid_model_88 5.994385937805915E8
## 5     30    500         0.8 rf_grid_model_96 5.998100993276515E8
##
## ---
##    mtries ntrees sample_rate       model_ids                 mse
## 91     30    200        0.55 rf_grid_model_6 6.584615189242806E8
## 92     24    300        0.55 rf_grid_model_9 6.605649622383039E8
## 93     20    200        0.55 rf_grid_model_1 6.607726014822638E8
## 94     28    200        0.55 rf_grid_model_5 6.616139369847372E8
## 95     24    200        0.55 rf_grid_model_3 6.635530575494395E8
## 96     22    200        0.55 rf_grid_model_2 6.729548370354824E8
```

La complessità temporale aumenta esponenzialmente per via del numero di combinazioni. Di conseguenza, h2o fornisce una tipologia alternativa di grid search chiamato *"RandomDiscrete"*, che salta da una combinazione casuale a un'altra e si ferma una volta rilevato un certo livello di miglioramento o una certa quantità di tempo è stata superata. Sebbene l'utilizzo di questa tecnica probabilmente non trovi il modello ottimale, in genere riesce a trovare un modello soddisfacente.

Nel codice seguente si esegue una grid search su 2.025 combinazioni di iperparametri. Con la random grid search che si fermerà se nessuno degli ultimi 10 modelli è riuscito a ottenere un miglioramento dello 0,5% del MSE rispetto al miglior modello precedente. Nel caso in cui si continuino ad ottenere miglioramenti, la ricerca si interrompe dopo 30 minuti. I risultati ottenuti dalla grid search hanno valutato 5 modelli e il modello migliore (max_depth= 35, min_rows= 1, mtries= 25, nbins= 10, ntrees= 500, sample_rate= .75) ha ottenuto un RMSE di 24608 ($\sqrt{6.056E8}$).

```r
# griglia di iperparametri
hyper_grid.h2o <- list(
  ntrees      = seq(200, 500, by = 150),
  mtries      = seq(15, 35, by = 10),
  max_depth   = seq(20, 40, by = 5),
  min_rows    = seq(1, 5, by = 2),
  nbins       = seq(10, 30, by = 5),
  sample_rate = c(.55, .632, .75)
)

# random grid search
search_criteria <- list(
  strategy = "RandomDiscrete",
  stopping_metric = "mse",
  stopping_tolerance = 0.005,
  stopping_rounds = 10,
  max_runtime_secs = 30*60
  )

# grid search
random_grid <- h2o.grid(
  algorithm = "randomForest",
```

```
  grid_id = "rf_grid2",
  x = x,
  y = y,
  training_frame = train.h2o,
  hyper_params = hyper_grid.h2o,
  search_criteria = search_criteria
  )

# collezione dei risultati e ordinamento per metrica di performance
grid_perf2 <- h2o.getGrid(
  grid_id = "rf_grid2",
  sort_by = "mse",
  decreasing = FALSE
  )
print(grid_perf2)

## H2O Grid Details
## ================
##
## Grid ID: rf_grid2
## Used hyper parameters:
##   -  max_depth
##   -  min_rows
##   -  mtries
##   -  nbins
##   -  ntrees
##   -  sample_rate
## Number of models: 140
## Number of failed models: 0
##
## Hyper-Parameter Search Summary: ordered by increasing mse
##   max_depth min_rows mtries nbins ntrees sample_rate          model_ids
## 1        35      1.0     25    10    500        0.75 rf_grid2_model_114
## 2        35      1.0     25    15    500        0.75   rf_grid2_model_6
## 3        40      1.0     15    25    350        0.75  rf_grid2_model_66
## 4        25      1.0     25    30    350        0.75  rf_grid2_model_29
## 5        40      1.0     15    10    500        0.75  rf_grid2_model_52
##                  mse
## 1 6.056085623294829E8
## 2 6.076645430909303E8
## 3  6.12618804864266E8
## 4 6.153110588111191E8
## 5 6.159044483611807E8
##
## ---
##     max_depth min_rows mtries nbins ntrees sample_rate          model_ids
## 135        25      5.0     15    30    500        0.55  rf_grid2_model_67
## 136        25      5.0     15    10    350        0.55 rf_grid2_model_112
## 137        40      5.0     15    15    350        0.55  rf_grid2_model_79
## 138        25      5.0     15    25    200        0.55 rf_grid2_model_128
## 139        30      5.0     15    20    200        0.55 rf_grid2_model_122
## 140        35      1.0     25    20    500        0.55 rf_grid2_model_140
##                   mse
## 135 7.336712686659595E8
## 136 7.346994279536405E8
## 137 7.368140144080462E8
## 138 7.421692020068938E8
## 139 7.467180178893355E8
## 140 8.446670748931651E8
```

Una volta identificato il modello migliore, è possibile applicarlo al set di test di per calcolare l'errore di test finale. Ne si deduce che è stato possibile ridurre l'RMSE a circa 24.000, riduzione di 10.000 rispetto al bagging.

```
# model_id per il miglior modello scelto dall'errore di validazione
best_model_id <- grid_perf2@model_ids[[1]]
best_model <- h2o.getModel(best_model_id)

# valutazione delle performance sul test set
ames_test.h2o <- as.h2o(ames_test)
best_model_perf <- h2o.performance(model = best_model, newdata = ames_test.h2o)

# RMSE del miglior modello
h2o.mse(best_model_perf) %>% sqrt()

## [1] 24189.3
```

Identificato il modello migliore, è possibile utilizzare la funzione *predict* per fare previsioni su un nuovo dataset.

```
# randomForest
pred_randomForest <- predict(ames_randomForest, ames_test)
head(pred_randomForest)

##        1        2        3        4        5        6
## 129459.5 185526.7 263271.6 196375.7 176455.2 392007.5

# ranger
pred_ranger <- predict(ames_ranger, ames_test)
head(pred_ranger$predictions)

## [1] 129130.5 186123.7 269912.0 198751.7 176939.0 395345.4

# h2o
pred_h2o <- predict(best_model, ames_test.h2o)
head(pred_h2o)

##    predict
## 1 128424.6
## 2 182748.2
## 3 265688.3
## 4 193110.2
## 5 176635.0
## 6 394836.4

h2o.shutdown(prompt=FALSE)
```

GRADIENT BOOSTING MACHINES

Esistono diversi pacchetti che implementano GBM e le sue varianti, nel presente caso di studio verranno utilizzati e confrontati *gbm e xgboost*.
Il pacchetto **gbm** include:

- GBM stocastico;
- Supporta la classificazione e gli alberi di regressione;
- Supporta diverse funzioni di perdita;
- Viene fornito uno stimatore out-of-bag per il numero ottimale di iterazioni;
- Facile incorrere in overfitting poiché la funzionalità di arresto anticipato non è automatizzata;
- Se viene utilizzata la cross validation interna, può essere parallelizzata a tutti i core sulla macchina;

Gbm ha due funzioni di training principali *gbm::gbm* e *gbm::gbm.fit*, la prima utilizza l'interfaccia della formula per specificare il modello mentre la seconda richiede matrici separate x e y (utile per lavorare con molte variabili). La configurazione standard di gbm include il learning rate (shrinkage) di 0,001, che è un tasso di apprendimento molto basso e in genere richiede un numero elevato di alberi per trovare l'MSE minimo, tuttavia gbm utilizza 100 alberi di standard (raramente sufficienti) che sono stati incrementati a 10.000. La profondità predefinita di ogni albero (*interaction.depth*) è 1, il che significa che stiamo assemblando un insieme di ceppi. Infine, viene utilizzato *cv.folds*

per eseguire una 5-fold cross validation. L'esecuzione del modello ha richiesto circa 90 secondi ed i risultati mostrano che la funzione di perdita MSE è ridotta al minimo con 9998 alberi.

```
# per riproducibilità
set.seed(123)

# train del modello GBM
gbm.fit <- gbm(
  formula = Sale_Price ~ .,
  distribution = "gaussian",
  data = ames_train,
  n.trees = 10000,
  interaction.depth = 1,
  shrinkage = 0.001,
  cv.folds = 5,
  n.cores = NULL,
  verbose = FALSE
  )

print(gbm.fit)
```

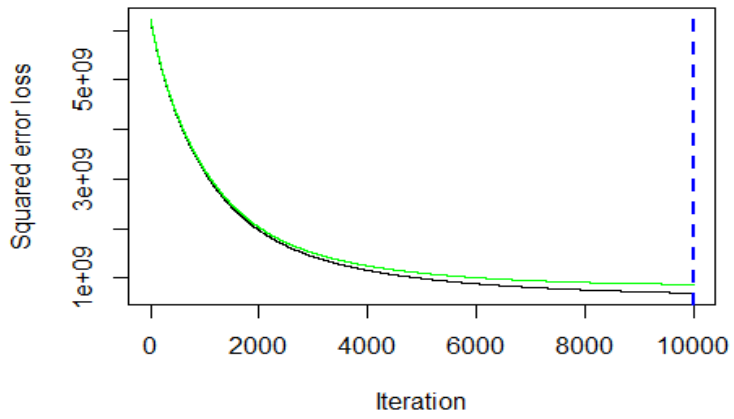
```
## gbm(formula = Sale_Price ~ ., distribution = "gaussian", data = ames_train,
##     n.trees = 10000, interaction.depth = 1, shrinkage = 0.001,
##     cv.folds = 5, verbose = FALSE, n.cores = NULL)
## A gradient boosted model with gaussian loss function.
## 10000 iterations were performed.
## The best cross-validation iteration was 9998.
## There were 80 predictors of which 47 had non-zero influence.
```

L'output è un elenco contenente diverse informazioni sulla modellazione e sui risultati a cui è possibile accedervi anche con un'indicizzazione regolare. Si noti che il CV RMSE minimo è 29551 (ciò significa che in media il modello raggiunge uno sconto di circa $ 29551 rispetto al prezzo di vendita effettivo) ma il grafico illustra anche che l'errore CV è in continua diminuzione anche a 10.000 alberi.

```
#  MSE e RMSE
sqrt(min(gbm.fit$cv.error))
```

```
## [1] 29551.99
```

```
# plot della funzione di perdita come risultato degli n alberi
gbm.perf(gbm.fit, method = "cv")
```



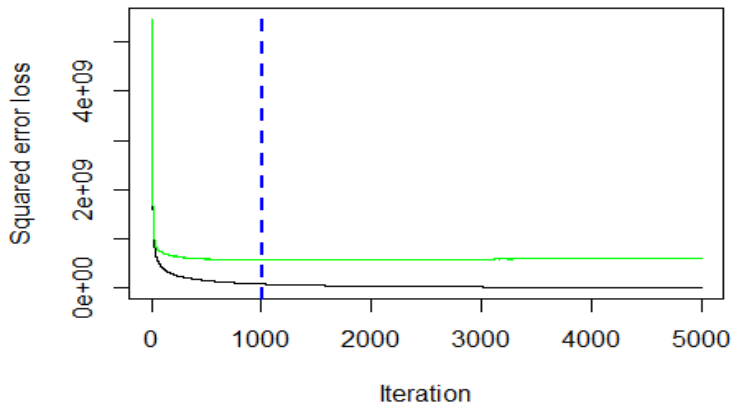

```
## [1] 9998
```

In questo caso, il basso tasso di apprendimento si traduce in piccoli miglioramenti incrementali, il che significa che sono necessari molti alberi.

Gbm - Tuning

Effettuando un tuning alternato dei parametri è possibile notare le differenze nei risultati. In primis si può aumentare il learning rate per effettuare passi più grandi lungo la gradient descent, ridurre il numero di alberi e aumentarne la profondità. Questo modello raggiunge un RMSE significativamente inferiore rispetto al modello iniziale con solo 1003 alberi.

```
# per riproducibilità
set.seed(123)

# train modello GBM
gbm.fit2 <- gbm(
  formula = Sale_Price ~ .,
  distribution = "gaussian",
  data = ames_train,
  n.trees = 5000,
  interaction.depth = 3,
  shrinkage = 0.1,
  cv.folds = 5,
  n.cores = NULL,
  verbose = FALSE
  )

# indice per gli n alberi con un errore minimo di cross validation
min_MSE <- which.min(gbm.fit2$cv.error)

#  MSE e RMSE
sqrt(gbm.fit2$cv.error[min_MSE])

## [1] 23852.61

# plot della funzione di perdita come risultato degli n alberi
gbm.perf(gbm.fit2, method = "cv")
```




```
## [1] 1003
```

Anche in questo caso è possibile configurare una grid search che itera su ogni combinazione di valori degli iperparametri, consentendone la valutazione. Si effettuerà una ricerca su 81 modelli con learning rate e profondità dell'albero variabili, variando anche il numero minimo di osservazioni consentite nei nodi terminali degli alberi (*n.minobsinnode*) e introducendo la stochastic gradient descent con *bag.fraction<1.*

```
# griglia di iperparametri
hyper_grid <- expand.grid(
  shrinkage = c(.01, .1, .3),
  interaction.depth = c(1, 3, 5),
  n.minobsinnode = c(5, 10, 15),
```

```
  bag.fraction = c(.65, .8, 1),
  optimal_trees = 0,
  min_RMSE = 0
)

# numero di combinazioni
nrow(hyper_grid)

## [1] 81
```

Si esegue un ciclo attraverso ciascuna combinazione di iperparametri definendo 5.000 alberi. Tuttavia, per accelerare il processo di tuning, invece di eseguire 5 volte la CV, si effettua il train sul 75% delle osservazioni di training e si valutano le prestazioni sul restante 25%. Il modello migliore ha prestazioni più elevate rispetto al a quello precedente, con l'RMSE di quasi $ 3.000 in meno. In secondo luogo, guardando i primi 10 modelli si nota che:

- Nessuno dei migliori modelli ha utilizzato un learning rate di 0,3;
- Nessuno dei migliori modelli ha utilizzato ceppi (interaction.depth = 1), ci sono probabilmente alcune importanti interazioni che gli alberi più profondi sono in grado di catturare;
- L'aggiunta di una componente stocastica con bag.fraction<1 sembra aiutare, potrebbero esserci dei minimi locali nella funzione di perdita;
- Quasi nessuno dei migliori modelli ha utilizzato n.minobsinnode= 15;
- In alcuni casi sembra che vengano utilizzati tutti i 5.000 alberi.

```
# randomizzare i dati
random_index <- sample(1:nrow(ames_train), nrow(ames_train))
random_ames_train <- ames_train[random_index, ]

# grid search
for(i in 1:nrow(hyper_grid)) {

  # riproducibilità
  set.seed(123)

  # train del modello
  gbm.tune <- gbm(
    formula = Sale_Price ~ .,
    distribution = "gaussian",
    data = random_ames_train,
    n.trees = 5000,
    interaction.depth = hyper_grid$interaction.depth[i],
    shrinkage = hyper_grid$shrinkage[i],
    n.minobsinnode = hyper_grid$n.minobsinnode[i],
    bag.fraction = hyper_grid$bag.fraction[i],
    train.fraction = .75,
    n.cores = NULL,
    verbose = FALSE
  )

  # aggiunta del min training error e degli alberi alla griglia
  hyper_grid$optimal_trees[i] <- which.min(gbm.tune$valid.error)
  hyper_grid$min_RMSE[i] <- sqrt(min(gbm.tune$valid.error))
}

hyper_grid %>%
  dplyr::arrange(min_RMSE) %>%
  head(10)

##    shrinkage interaction.depth n.minobsinnode bag.fraction optimal_trees
## 1       0.10                 5              5         0.80           583
## 2       0.01                 5              5         0.80          4969
## 3       0.01                 5              5         1.00          4844
```

```
## 4       0.01                 5              10         0.80          4669
## 5       0.01                 5              10         1.00          4417
## 6       0.10                 5              15         0.80           364
## 7       0.01                 5              10         0.65          5000
## 8       0.01                 5               5         0.65          5000
## 9       0.10                 5              10         0.80          1417
## 10      0.01                 5              15         0.80          5000
##    min_RMSE
## 1  20916.08
## 2  21341.28
## 3  21402.16
## 4  21418.73
## 5  21496.43
## 6  21527.81
## 7  21569.37
## 8  21599.52
## 9  21652.69
## 10 21694.30
```

Questi risultati aiutano a definire le aree in cui affinare la ricerca sulla griglia, considerando le regioni più vicine ai valori che sembrano produrre i migliori risultati.

```
# modifica della griglia di iperparametri
hyper_grid <- expand.grid(
  shrinkage = c(.01, .05, .1),
  interaction.depth = c(3, 5, 7),
  n.minobsinnode = c(5, 7, 10),
  bag.fraction = c(.65, .8, 1),
  optimal_trees = 0,
  min_RMSE = 0
)

# numero totale di combinazioni
nrow(hyper_grid)

## [1] 81
```

E' possibile usare lo stesso ciclo for di prima ed eseguire la grid search, ottenendo di poco migliori rispetto ai precedenti.

```
# grid search
for(i in 1:nrow(hyper_grid)) {

  # riproducibilità
  set.seed(123)

  # train del modello
  gbm.tune <- gbm(
    formula = Sale_Price ~ .,
    distribution = "gaussian",
    data = random_ames_train,
    n.trees = 6000,
    interaction.depth = hyper_grid$interaction.depth[i],
    shrinkage = hyper_grid$shrinkage[i],
    n.minobsinnode = hyper_grid$n.minobsinnode[i],
    bag.fraction = hyper_grid$bag.fraction[i],
    train.fraction = .75,
    n.cores = NULL,
    verbose = FALSE
  )

  # ggiunta del min training error e degli alberi alla griglia
```

```
  hyper_grid$optimal_trees[i] <- which.min(gbm.tune$valid.error)
  hyper_grid$min_RMSE[i] <- sqrt(min(gbm.tune$valid.error))
}

hyper_grid %>%
  dplyr::arrange(min_RMSE) %>%
  head(10)

##    shrinkage interaction.depth n.minobsinnode bag.fraction optimal_trees
## 1       0.05                 5              7         0.65          1088
## 2       0.10                 5              5         0.80           583
## 3       0.05                 7              7         0.65          1624
## 4       0.01                 7              5         1.00          5727
## 5       0.05                 5              5         0.65          5247
## 6       0.05                 7              7         1.00          1160
## 7       0.05                 7              5         1.00          1196
## 8       0.01                 7              5         0.80          5979
## 9       0.01                 7              7         0.80          5409
## 10      0.05                 7              5         0.80          2432
##    min_RMSE
## 1  20818.68
## 2  20916.08
## 3  21007.13
## 4  21042.62
## 5  21047.53
## 6  21100.00
## 7  21140.22
## 8  21164.36
## 9  21174.42
## 10 21233.68
```

Definito il miglior modello si effettua il suo training.

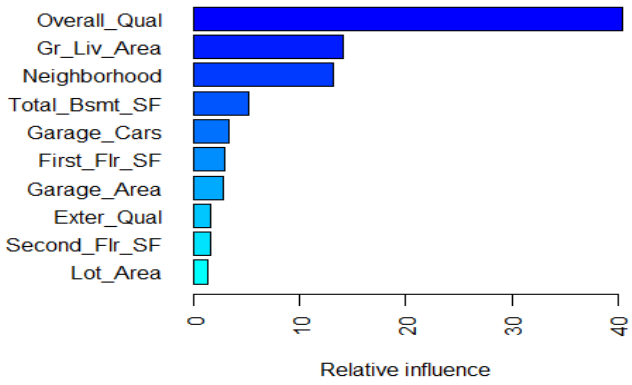
```
# riproducibilità
set.seed(123)

# train modello GBM
gbm.fit.final <- gbm(
  formula = Sale_Price ~ .,
  distribution = "gaussian",
  data = ames_train,
  n.trees = 1088,
  interaction.depth = 5,
  shrinkage = 0.5,
  n.minobsinnode = 7,
  bag.fraction = .65,
  train.fraction = 1,
  n.cores = NULL,
  verbose = FALSE
  )
```

Gbm - Visualization

Tramite il metodo *summary* è possibile comprendere quali siano le variabili che hanno la maggiore influenza sul prezzo di vendita.

1. *method = relative.influence:* Ad ogni split in ogni albero, gbm calcola il miglioramento del MSE e ricava la media del miglioramento apportato da ciascuna variabile in tutti gli alberi in cui è utilizzata;
2. *method = permutation.test.gbm*: Per ogni albero, il campione OOB viene trasmesso e viene registrata l'accuratezza della previsione. Successivamente vengono permutati casualmente i valori per ciascuna variabile e la precisione viene nuovamente calcolata. La diminuzione della precisione viene mediata su tutti gli alberi per ciascuna variabile e le variabili

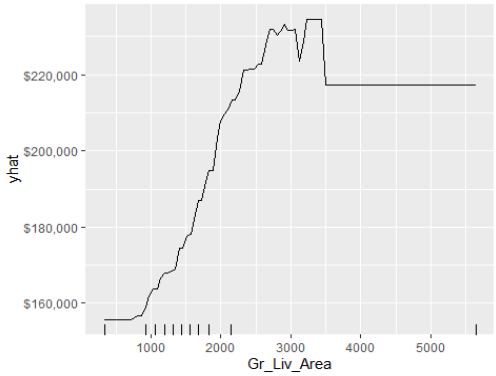
```r
par(mar = c(5, 8, 1, 1))
summary(
  gbm.fit.final,
  cBars = 10,
  method = relative.influence,
  las = 2
  )
```




```
##                                  var      rel.inf
## Overall_Qual             Overall_Qual 4.048861e+01
## Gr_Liv_Area               Gr_Liv_Area 1.402976e+01
## Neighborhood             Neighborhood 1.320178e+01
## Total_Bsmt_SF           Total_Bsmt_SF 5.227608e+00
## Garage_Cars               Garage_Cars 3.296480e+00
## First_Flr_SF             First_Flr_SF 2.927688e+00
## Garage_Area               Garage_Area 2.779194e+00
## Exter_Qual                 Exter_Qual 1.618451e+00
## Second_Flr_SF           Second_Flr_SF 1.517907e+00
## Lot_Area                     Lot_Area 1.296325e+00
```

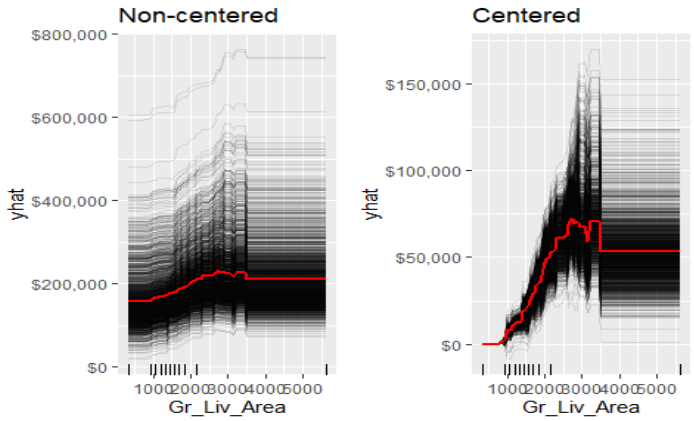
Dopo aver identificato le variabili più rilevanti, il passo successivo è tentare di capire come la variabile di risposta cambia in relazione ad esse. Per questo è possibile usare i **partial dependence plots(PDP)** e le **individual conditional expectation curves(ICE)**. I PDP tracciano la variazione del valore medio previsto quando le features specificate variano sulla loro distribuzione marginale. Considerando la variabile Gr_Liv_Area, il grafico PDP mostra la variazione media del prezzo di vendita previsto al variare Gr_Liv_Area mantenendo costanti tutte le altre variabili. Questo PDP illustra come il prezzo di vendita previsto aumenta all'aumentare della metratura del piano terra di una casa.

```r
gbm.fit.final %>%
  partial(pred.var = "Gr_Liv_Area", n.trees = gbm.fit.final$n.trees, grid.resolution = 100) %>%
  autoplot(rug = TRUE, train = ames_train) +
  scale_y_continuous(labels = scales::dollar)
```



Le curve ICE sono un'estensione dei grafici PDP ma, piuttosto che tracciare l'effetto marginale medio sulla variabile di risposta, tracciano la variazione nella variabile di risposta prevista per ogni osservazione al variare della variabile predittore. Quando le curve risultano "impilate" l'una sull'altra, può essere difficile distinguere l'eterogeneità nei valori della variabile di risposta per via di variazioni marginali nella variabile predittiva di interesse. L'ICE centrato può aiutare a trarre queste inferenze e può evidenziare qualsiasi tipo di eterogeneità nei risultati. I risultati mostrano che la maggior parte delle osservazioni segue una tendenza comune all'aumentare di *Gr_Liv_Area*, tuttavia, il grafico ICE centrato evidenzia alcune osservazioni che si discostano dalla tendenza comune.

```
ice1 <- gbm.fit.final %>%
  partial(
    pred.var = "Gr_Liv_Area",
    n.trees = gbm.fit.final$n.trees,
    grid.resolution = 100,
    ice = TRUE
    ) %>%
  autoplot(rug = TRUE, train = ames_train, alpha = .1) +
  ggtitle("Non-centered") +
  scale_y_continuous(labels = scales::dollar)

ice2 <- gbm.fit.final %>%
  partial(
    pred.var = "Gr_Liv_Area",
    n.trees = gbm.fit.final$n.trees,
    grid.resolution = 100,
    ice = TRUE
    ) %>%
  autoplot(rug = TRUE, train = ames_train, alpha = .1, center = TRUE) +
  ggtitle("Centered") +
  scale_y_continuous(labels = scales::dollar)

gridExtra::grid.arrange(ice1, ice2, nrow = 1)
```

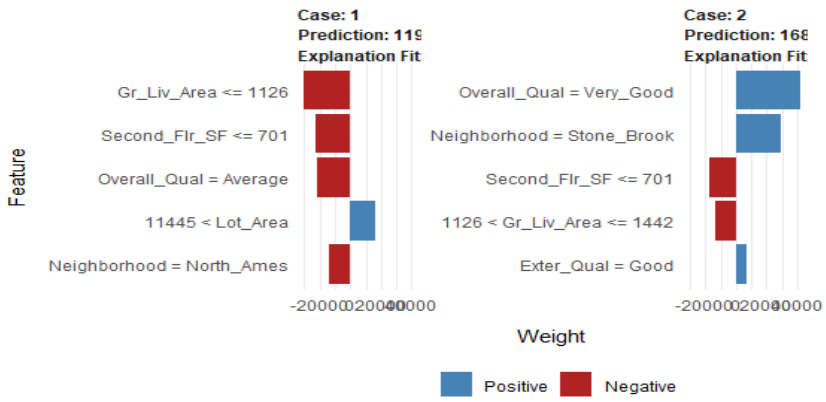


LIME è una procedura più recente che aiuta a comprendere perché una predizione ha prodotto un dato valore per una singola osservazione.

```
model_type.gbm <- function(x, ...) {
  return("regression")
}

predict_model.gbm <- function(x, newdata, ...) {
  pred <- predict(x, newdata, n.trees = x$n.trees)
  return(as.data.frame(pred))
}
```

I risultati mostrano il valore previsto, il fit del modello locale e le variabili più influenti che determinano il valore previsto per ciascuna osservazione.

```
# usare poche osservazioni
local_obs <- ames_test[1:2, ]

# LIME
explainer <- lime(ames_train, gbm.fit.final)
explanation <- explain(local_obs, explainer, n_features = 5)
plot_features(explanation)
```



Gbm - Predicting

Come per la maggior parte dei modelli, è possibile usare semplicemente la funzione *predict*. Si noti che l'RMSE per il test set è molto vicino all'RMSE ottenuto con il miglior modello gbm.

```
# predict values for test data
pred <- predict(gbm.fit.final, n.trees = gbm.fit.final$n.trees, ames_test)

# results
caret::RMSE(pred, ames_test$Sale_Price)

## [1] 21533.65
```

Il pacchetto **xgboost** fornisce un'API R per "Extreme Gradient Boosting", che è un'implementazione efficiente del framework di gradient boosting.
Feature principali:

- **Parallelizzazione**: implementa gli alberi sequenzialmente ed in maniera parallela grazie alla natura dei cicli utilizzati per definire i base learners, il ciclo esterno enumera i nodi foglia di un albero mentre il secondo (interno) calcola le features. Questo annidamento limita la parallelizzazione perchè senza completare il cilo interno, quello esterno non può essere avviato. Pertanto, per migliorare i tempi di esecuzione, l'ordine dei cicli viene scambiato utilizzando l'inizializzazione tramite una scansione globale di tutte le istanze e l'ordinamento utilizzando thread paralleli.
- **Potatura degli alberi**: viene utilizzato *max_depth* per potare gli alberi, migliorando significativamente le prestazioni di calcolo;
- **Ottimizzazione hardware**: l'utilizzo della cache viene ottimizzato allocando i buffer interni in ogni thread per memorizzare le statistiche del gradiente;
- **Regolarizzazione**: Vengono penalizzati modelli più complessi per evitare overfitting;
- **Sparsità**: gestione migliorata delle features sparse tramite un apprendimento dei valori mancanti sulla base della perdita sul training;
- **Cross-validation**: metodo CV built-in ad ogni iterazione.

XGBoost funziona con matrici contenenti solo variabili numeriche, di conseguenza, occorre codificare i dati. In questo caso è stato usato usato vtreat, un pacchetto per la preparazione dei dati che aiuta a eliminare i problemi causati da valori mancanti.

Si utilizza vtreat per effettuare una codifica one-hot dei dataset di train e test.

```
# nomi delle variabili
features <- setdiff(names(ames_train), "Sale_Price")

# creazione del piano di treatment dai dati di training
treatplan <- vtreat::designTreatmentsZ(ames_train, features, verbose = FALSE)

# nomi di variabili pulite da scoreFrame
new_vars <- treatplan %>%
  magrittr::use_series(scoreFrame) %>%
  dplyr::filter(code %in% c("clean", "lev")) %>%
  magrittr::use_series(varName)

# preparazione dei dati di training
features_train <- vtreat::prepare(treatplan, ames_train, varRestriction = new_vars) %>% as.matrix()
response_train <- ames_train$Sale_Price

# preparazione dei dati di test
features_test <- vtreat::prepare(treatplan, ames_test, varRestriction = new_vars) %>% as.matrix()
response_test <- ames_test$Sale_Price

# dimensione dei dati one hot
dim(features_train)

## [1] 2051  343

dim(features_test)

## [1] 879 343
```

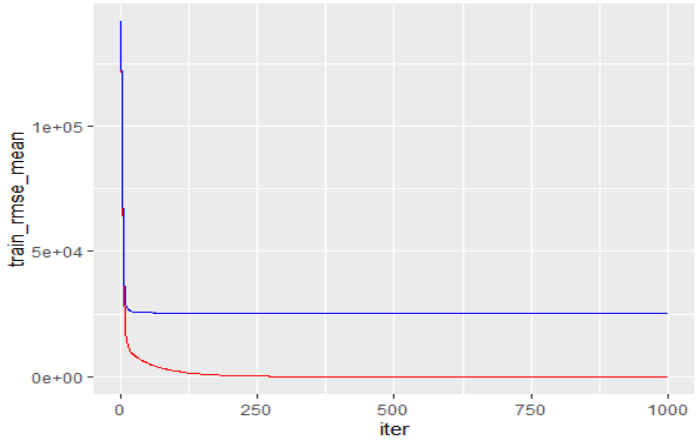
xgboost fornisce diverse funzioni di training come *xgb.cv*, che incorpora la cross validation. Quanto segue addestra un modello XGBoost con una 5 fold cross validation, con 1.000 alberi e i restanti valori predefiniti:

- Learning rate (*eta*): 0,3
- Profondità albero (*max_depth*): 6
- Dimensione minima del nodo (*min_child_weight*): 1
- Percentuale di dati di training da campionare per ogni albero: 100%

```
# reproducibility
set.seed(123)

xgb.fit1 <- xgb.cv(
  data = features_train,
  label = response_train,
  nrounds = 1000,
  nfold = 5,
  objective = "reg:linear",
  verbose = 0
)
```

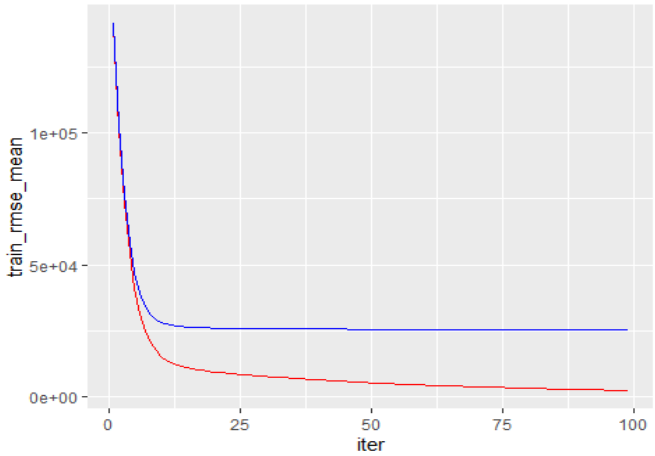
L' oggetto *xgb.fit1* contiene molte informazioni, in particolare si può valutare l' *xgb.fit1$evaluation_log* per identificare l'RMSE minimo e il numero ottimale di alberi sia per i dati di training che per l'errore di cross-validaton. Si può notare che l'errore di training continua a diminuire fino a 994 alberi dove l'RMSE raggiunge quasi lo zero, tuttavia, l'errore con cross-validation raggiunge un RMSE minimo di $ 25.435 con solo 161 alberi.

```r
# numero di alberi per minimizzare l'errore
xgb.fit1$evaluation_log %>%
  dplyr::summarise(
    ntrees.train = which(train_rmse_mean == min(train_rmse_mean))[1],
    rmse.train   = min(train_rmse_mean),
    ntrees.test  = which(test_rmse_mean == min(test_rmse_mean))[1],
    rmse.test   = min(test_rmse_mean)
  )

##   ntrees.train rmse.train ntrees.test rmse.test
## 1          994  0.0488872         161  25435.18

# plot errore vs numero di alberi
ggplot(xgb.fit1$evaluation_log) +
  geom_line(aes(iter, train_rmse_mean), color = "red") +
  geom_line(aes(iter, test_rmse_mean), color = "blue")
```



Un'altra funzionalità fornita da *xgb.cv* è l'arresto anticipato, che permette di interrompere l'esecuzione se l'errore di cross validation non migliora per n alberi continui.

```r
# riproducibilità
set.seed(123)

xgb.fit2 <- xgb.cv(
  data = features_train,
  label = response_train,
  nrounds = 1000,
  nfold = 5,
  objective = "reg:linear",  # modelli di regressione
  verbose = 0,
  early_stopping_rounds = 10 # stop se non c'è miglioramento per 10 alberi consecutivi
)

# plot errore vs numero di alberi
ggplot(xgb.fit2$evaluation_log) +
  geom_line(aes(iter, train_rmse_mean), color = "red") +
  geom_line(aes(iter, test_rmse_mean), color = "blue")
```



XGBoost - Tuning

I parametri più comuni di XGBoost includono:

- *eta*: controlla il learning rate
- *max_depth*: profondità dell'albero
- *min_child_weight*: numero minimo di osservazioni richieste in ogni nodo terminale
- *subsample*: percentuale di dati di training da campionare per ogni albero
- *colsample_bytrees*: percentuale di colonne da cui campionare per ogni albero

Si può estendere il modello precedente con i seguenti parametri.

```
# lista di parametri
  params <- list(
    eta = .1,
    max_depth = 5,
    min_child_weight = 2,
    subsample = .8,
    colsample_bytree = .9
  )

# riproducibilità
set.seed(123)

# train del modello
xgb.fit3 <- xgb.cv(
  params = params,
  data = features_train,
  label = response_train,
  nrounds = 1000,
  nfold = 5,
  objective = "reg:linear",
  verbose = 0,
  early_stopping_rounds = 10
)

# valutazione dei risultati
xgb.fit3$evaluation_log %>%
  dplyr::summarise(
    ntrees.train = which(train_rmse_mean == min(train_rmse_mean))[1],
    rmse.train   = min(train_rmse_mean),
    ntrees.test  = which(test_rmse_mean == min(test_rmse_mean))[1],
    rmse.test   = min(test_rmse_mean)
  )
```

```
##   ntrees.train rmse.train ntrees.test rmse.test
## 1          178   5974.205         168  23230.09
```

Si può eseguire una grid search creando la griglia iperparametrica.

```
# griglia iperparametrica
hyper_grid <- expand.grid(
  eta = c(.01, .05, .1, .3),
  max_depth = c(1, 3, 5, 7),
  min_child_weight = c(1, 3, 5, 7),
  subsample = c(.65, .8, 1),
  colsample_bytree = c(.8, .9, 1),
  optimal_trees = 0,               # dump dei risultati
  min_RMSE = 0                     # dump dei risultati
)

nrow(hyper_grid)

## [1] 576
```

Si applica un loop per applicare il modello XGBoost a ciascuna combinazione di iperparametri e caricare i risultati nel dataframe *hyper_grid*.

```
# grid search
for(i in 1:nrow(hyper_grid)) {

  # creazione lista di parametri
  params <- list(
    eta = hyper_grid$eta[i],
    max_depth = hyper_grid$max_depth[i],
    min_child_weight = hyper_grid$min_child_weight[i],
    subsample = hyper_grid$subsample[i],
    colsample_bytree = hyper_grid$colsample_bytree[i]
  )

  # riproducibilità
  set.seed(123)

  # train del modello
  xgb.tune <- xgb.cv(
    params = params,
    data = features_train,
    label = response_train,
    nrounds = 5000,
    nfold = 5,
    objective = "reg:linear",
    verbose = 0,
    early_stopping_rounds = 10
  )

  # aggiunta del min training error e degli alberi alla griglia
  hyper_grid$optimal_trees[i] <- which.min(xgb.tune$evaluation_log$test_rmse_mean)
  hyper_grid$min_RMSE[i] <- min(xgb.tune$evaluation_log$test_rmse_mean)
}

hyper_grid %>%
  dplyr::arrange(min_RMSE) %>%
  head(10)

##     eta max_depth min_child_weight subsample colsample_bytree
## 1  0.01         5                1      0.65              1.0
## 2  0.05         5                1      0.65              0.9
```

```
## 3  0.05         5                1      0.65              1.0
## 4  0.01         5                1      0.65              0.8
## 5  0.01         5                3      0.65              0.9
## 6  0.05         5                3      0.80              0.8
## 7  0.01         5                3      0.65              0.8
## 8  0.05         5                7      0.65              0.9
## 9  0.01         5                3      0.80              1.0
## 10 0.01         5                3      0.65              1.0
##    optimal_trees min_RMSE
## 1           1463 22163.11
## 2            346 22252.92
## 3            338 22253.13
## 4           1513 22262.25
## 5           1713 22276.96
## 6            352 22281.78
## 7           1467 22286.62
## 8            458 22298.54
## 9           1695 22301.45
## 10          1425 22306.67
```

Una volta trovato il modello ottimale, è possibile adattare il modello finale *xgb.train*.

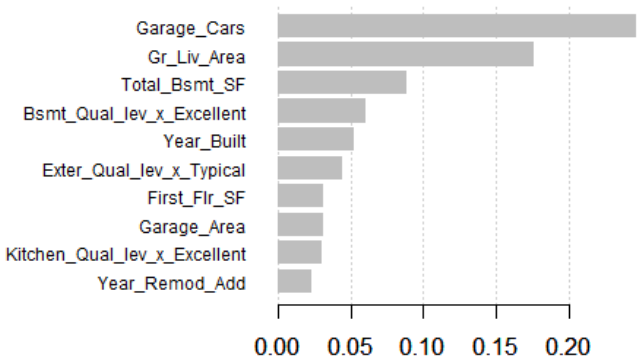
```
# lista di parametri
params <- list(
  eta = 0.01,
  max_depth = 5,
  min_child_weight = 1,
  subsample = 0.65,
  colsample_bytree = 1
)

# train del modello finale
xgb.fit.final <- xgboost(
  params = params,
  data = features_train,
  label = response_train,
  nrounds = 1463,
  objective = "reg:linear",
  verbose = 0
)
```

XGBoost - Visualization

Xgboost fornisce grafici built-in per visualizzare l'importanza delle variabili. Per fare ciò è necessario creare una matrice di importanza con *xgb.importance* ed inserirla in *xgb.plot.importance*.
Sono disponibili 3 misure di importanza per le variabili:

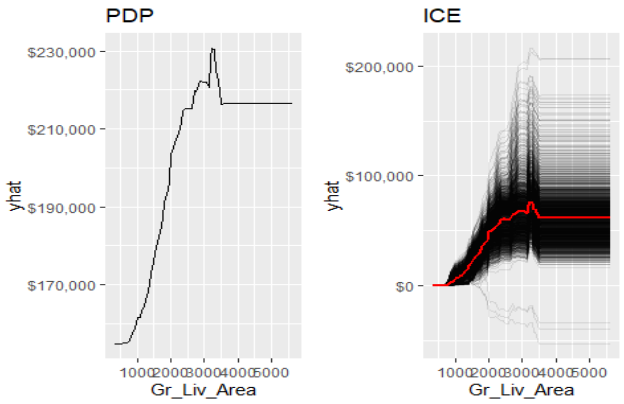
- *Gain*: il contributo relativo della feature corrispondente al modello calcolato considerando il contributo di ciascuna feature per ogni albero nel modello.
- *Cover*: il numero relativo di osservazioni per la feature;
- *Frequency*: la percentuale che rappresenta il numero relativo di volte in cui una particolare feature si verifica negli alberi del modello.

```
# creazione matrice di importanza
importance_matrix <- xgb.importance(model = xgb.fit.final)

# plot
xgb.plot.importance(importance_matrix, top_n = 10, measure = "Gain")
```



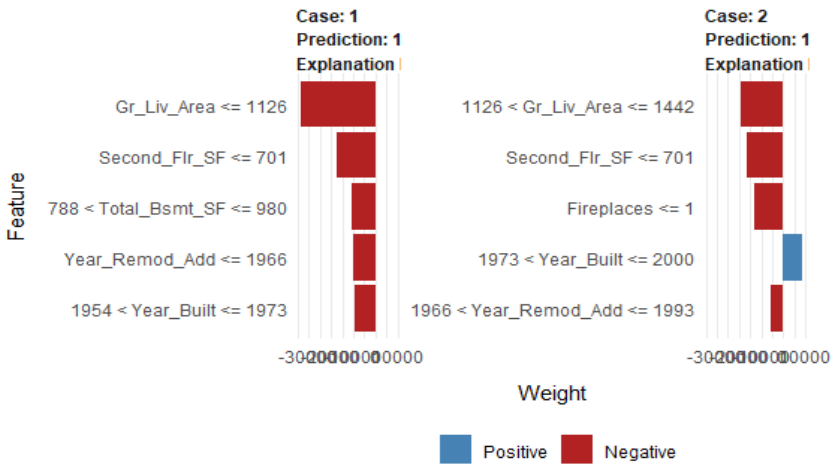
I grafici PDP e ICE funzionano in modo simile a gbm, l'unica differenza è che occorre incorporare i dati di training all'interno della funzione *partial*.

```
pdp <- xgb.fit.final %>%
  partial(pred.var = "Gr_Liv_Area", n.trees = 1576, grid.resolution = 100, train = features_train)
%>%
  autoplot(rug = TRUE, train = features_train) +
  scale_y_continuous(labels = scales::dollar) +
  ggtitle("PDP")

ice <- xgb.fit.final %>%
  partial(pred.var = "Gr_Liv_Area", n.trees = 1576, grid.resolution = 100, train = features_train,
ice = TRUE) %>%
  autoplot(rug = TRUE, train = features_train, alpha = .1, center = TRUE) +
  scale_y_continuous(labels = scales::dollar) +
  ggtitle("ICE")

gridExtra::grid.arrange(pdp, ice, nrow = 1)
```



LIME fornisce funzionalità integrate per gli oggetti xgboost.

```
# one-hot encode delle osservazioni locali per essere valutate
local_obs_onehot <- vtreat::prepare(treatplan, local_obs, varRestriction = new_vars)

# LIME
explainer <- lime(data.frame(features_train), xgb.fit.final)
explanation <- explain(local_obs_onehot, explainer, n_features = 5)
plot_features(explanation)
```



XGBooost - Predicting

Infine, si usa *predict* per effettuare predizioni su nuove osservazioni; tuttavia, a differenza di gbm non è necessario fornire il numero di alberi. Il set di test RMSE è solo di circa $ 100 diverso da quello prodotto dal modello gbm.

```
# predizione dei valori per i dati di test
pred <- predict(xgb.fit.final, features_test)

# risultati
caret::RMSE(pred, response_test)

## [1] 21657.96
```

In conclusione si può dunque affermare che le implementazioni del boosting tramite le due librerie considerate restituiscono risultati simili tra loro, sensibilmente migliori rispetto a quelli ottenuti con l'implementazione di Random Forest e, di conseguenza, nettamente preferibili a quelli ottenuti tramite il semplice bagging.