UNIVERSITÀ DEGLI STUDI DI UDINE

DIPARTIMENTO POLITECNICO DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in ***Ingegneria Gestionale***

PROGETTO PER IL CORSO APPLIED STATISTICS

Pacchetti tree e rpart per alberi decisionali

Nicolò Della Bianca

Data di compilazione: 2024-05-09

Abstract

Il presente elaborato tratterà l'applicazione dei pacchetti *tree* ed *rpart* disponibili in R per la generazione di alberi decisionali.

Gli alberi decisionali sono un metodo di classificazione che si basa sulla suddivisione iterativa degli oggetti, allo scopo di assegnarli a un numero finito o potenzialmente infinito di classi.

Questi alberi sono costituiti da nodi interni che verificano le condizioni sugli attributi di input e diramano il flusso di dati a seconda dei risultati. Ci sono due tipi di alberi decisionali: quelli di regressione e quelli di classificazione. Gli alberi di regressione prevedono valori continui e quelli di classificazione prevedono variabili categoriche.

Gli alberi decisionali sono facili da interpretare e possono gestire dati non preparati. Tuttavia, ci sono anche svantaggi come l'overfitting, l'elevata varianza e i costi di computazione elevati. Per migliorare la precisione, è possibile combinare più alberi decisionali con tecniche come il "*bagging*", le "*random forest*" e il "*boosting*".

Successivamente, verrà approfondito l'utilizzo del pacchetto tree e del pacchetto rpart disponibili in R. Inoltre, attraverso un'analisi delle principali funzioni e relativi parametri, verrà proposto un confronto qualitativo dei due pacchetti.

Infine, verranno presentati degli esempi applicativi sia per alberi di regressione che per alberi di classificazione, utilizzando due dataset diversi. Per la regressione sarà valutata la resistenza del calcestruzzo, mentre per la classificazione sarà catalogata la propensione al pagamento dei clienti richiedenti un finanziamento.

Indice

1 Alberi Decisionali **4**

1.1 Tipologie di alberi decisionali 5

1.1.1 Alberi di regressione 5

1.1.2 Alberi di classificazione 6

1.2 Vantaggi e svantaggi 6

1.3 Bagging, Random Forest, Boosting 7

1.3.1 Bagging 7

1.3.2 Random Forest 9

1.3.3 Boosting 10

2 Utilizzo e confronto pacchetti **11**

2.1 Il pacchetto tree 11

2.1.1 Funzioni principali 12

2.1.2 Altre funzioni 17

2.2 Il pacchetto rpart 17

2.2.1 Funzioni principali 18

2.2.2 Altre funzioni 23

2.3 Confronto 24

3 Esempi applicativi **25**

3.1 Classificazione 25

3.1.1 Training 25

3.1.2 Grafici 26

3.1.3 Test 27

3.1.4 Cross-validation 28

3.1.5 Potatura 29

3.1.6 Test 30

3.2 Regressione 31

3.2.1 Training 31

3.2.2 Grafici 32

3.2.3 Cross-validation 32

3.2.4 Potatura 34

3.2.5 Test 34

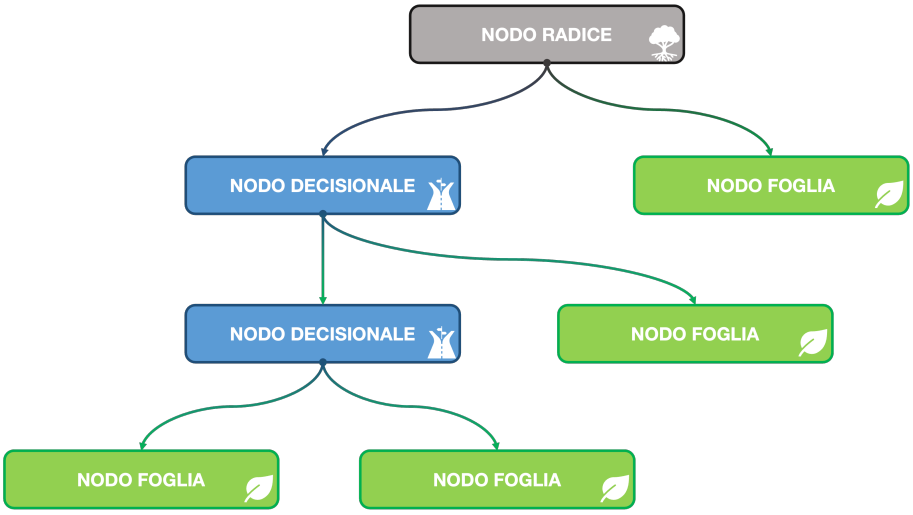
3.3 Confronto 36

4 Conclusioni **37**

5 Sitografia **38**

1 Alberi Decisionali

Gli alberi di decisionali costituiscono il modo più semplice di classificare degli oggetti in un finito, ma, potenzialmente, anche infinito numeri di classi. Di fatto, dunque, un albero decisionale è un algoritmo di apprendimento induttivo, basato sull'osservazione del contesto circostante e viene utilizzato per problemi di classificazione e di regressione (o previsione). Si tratta di un sistema con *n* variabili in input e *m* variabili in output, le quali possono essere categoriche o continue.
L'albero decisionale è una struttura simile a un diagramma di flusso, in cui il processo è rappresentato con un albero logico rovesciato che comincia sempre dal nodo radice, ossia il nodo che si trova più in alto nella struttura, e prosegue verso il basso. Osservando la figura, ogni nodo interno verifica una condizione (test) su un attributo e a seconda dei valori rilevati, ogni ramo indirizza il flusso verso una direzione oppure un'altra, rappresentando il risultato del test. Infine, man mano che il processo di selezione prosegue, lo spazio di ipotesi si riduce finché ogni ramo decisionale non viene eliminato e si raggiunge il nodo foglia (nodo terminale) contenente la decisione finale.



Le proprietà più importanti degli alberi decisionali, osservando l'immagine qui sopra, sono le seguenti:

1. Il nodo radice rappresenta l'intero campione ed è suddiviso in due o più insiemi omogenei;
2. La suddivisione è un processo di divisione di un nodo in due o più sotto-nodi;
3. Quando un sotto-nodo si divide in altri sotto-nodi, viene chiamato nodo decisionale;
4. I nodi che non si dividono sono chiamati nodi terminali o foglie;
5. La potatura è quel processo per cui vengono rimossi i sotto-nodi di un nodo decisionale. L'opposto della potatura è la divisione;
6. Una sottosezione di un intero albero è chiamata ramo;
7. Un nodo diviso in sotto-nodi è chiamato nodo genitore dei sotto-nodi, mentre i sotto-nodi sono chiamati figli del nodo genitore.

1.1 Tipologie di alberi decisionali

Le variabili di un albero decisionale possono essere:

- Variabili continue: variabili con valori numerici reali, utilizzate come variabili di risposta negli alberi di regressione;
- Variabili discrete: variabili categoriche, utilizzate come variabili di risposta negli alberi di classificazione.

Gli alberi di regressione e di classificazione sono delle metodologie molto diverse dai classici metodi di regressione. Sebbene per insiemi di dati di grandi dimensioni possano rivelare strutture complesse, sono delle metodologie molto semplici ed intuitive e possono essere applicate ad un vasto insieme di contesti.

1.1.1 Alberi di regressione

La regressione ha l'obiettivo di stimare la relazione tra una variabile di risposta ed un insieme di variabili esplicative. Nel caso degli alberi di regressione, le variabili sono rappresentate dai nodi interni, i possibili valori sono i rami e, infine, i nodi terminali sono coloro che descrivono il valore predetto durante il cammino dal nodo genitore al nodo foglia.

Un albero di regressione può fungere da strumento di previsione, aiutare a determinare la decisione migliore, identificare i possibili eventi che potrebbero verificarsi e vederne i potenziali risultati. L'analisi viene realizzata mediante l'utilizzo di tecniche di apprendimento a partire dall'insieme dei dati iniziali (training set).

Questi modelli permettono di dividere i dati in sottoinsiemi, poiché selezionano le suddivisioni che riducono la dispersione dei valori degli attributi target e, pertanto, essi possono essere stimati dai loro valori medi nei nodi terminali. Ogni nodo è diviso in due o più nodi foglia mediante la selezione del fattore predittivo che riduce maggiormente la somma dei quadrati dei residui di tale nodo. Si sfrutta l'utilizzo della media e della deviazione standard, le quali, assieme al numero di righe di dati corrispondenti al nodo, sono direttamente correlate alla somma dei quadrati dei residui.

Talvolta, quando la somma dei quadrati dei residui è tollerabile entro un certo limite, i nodi terminali corrispondenti a determinate categorie di fattori predittivi vengono uniti. L'utilizzo di R^2 è utile per stimare la forza predittiva dell'albero di regressione: se essa è superiore alla soglia del 10%, l'albero è considerato affidabile.

Il vantaggio principale degli alberi di regressione è la loro leggibilità, in quanto, oltre a prevedere i valori degli attributi target, spiegano anche quali attributi vengono utilizzati ed in che modo il loro impiego genera le previsioni ottenute.

Per creare e utilizzare i modelli degli alberi di regressione è necessario seguire le attività di:

- Crescita;
- Potatura;
- Previsione.

In particolare, la crescita si effettua creando un albero di regressione a partire da un dataset. Vengono selezionate le suddivisioni, per ridurre la dispersione degli attributi target ed in contemporanea vengono assegnati i valori target alle foglie quando non sono necessarie o possibili ulteriori divisioni.

La potatura viene realizzata per ridurre il rischio di overfitting. A tal fine, i nodi che non migliorano la qualità di previsione attesa sui nuovi dati vengono sostituiti dalle foglie.

Infine, la previsione utilizza un albero di regressione già analizzato per prevedere i valori degli attributi target del dataset preso in considerazione.

1.1.2 Alberi di classificazione

Gli alberi di classificazione sono simili agli alberi di regressione, ad eccezione del fatto che vengono utilizzati per la previsione di risultati qualitativi piuttosto che quantitativi. Difatti, questa tipologia di albero viene generata quando le variabili target sono categoriche. Come per gli alberi di regressione, per far "crescere" un albero di classificazione si utilizza la suddivisione binaria ricorsiva. Tuttavia, nell'impostazione della classificazione, la somma dei quadrati dei residui non può essere utilizzata come criterio di suddivisione, bensì si applica uno dei seguenti tre metodi:

- *Tasso di errore di classificazione (misclass)*: non si osserva quanto una risposta numerica si allontana dal valore medio, come nell'impostazione della regressione, bensì si definisce il "tasso di errore" per determinare la frazione di osservazioni del training set in una regione che non appartiene alla classe più diffusa;
- *Impurità di Gin*i: ogni nodo viene diviso in due o più nodi foglia, riducendo man mano il valore della misura di impurità di Gini del nodo. L'impurità di Gini corrisponde ad una metrica di errore alternativa, progettata per dimostrare quanti dei dati del training set di una specifica regione appartengono ad una singola classe, ovvero quanto "pura" sia la regione. Si ottengono piccoli valori dell'impurità di Gini quando una regione contiene dati provenienti da un'unica classe. Inoltre, per ogni nodo, il fattore predittivo che riduce maggiormente il valore dell'impurità di Gini viene scelto per dividere il nodo;
- *Entropia incrociata*: funziona in maniera simile all'impurità di Gini, in quanto vengono generati piccoli valori di entropia incrociata quando il nodo è "puro".

Durante la generazione di un albero di classificazione, l'entropia incrociata e l'impurità di Gini sono i criteri più utilizzati per valutare la qualità delle suddivisioni, siccome sono più sensitivi alla purità dei nodi rispetto al tasso di errore di classificazione. Tuttavia, ognuno di questi tre metodi possono, invece, essere applicati per la potatura dell'albero, ma più comunemente è preferito il tasso di errore di classificazione.

1.2 Vantaggi e svantaggi

Nella sezione precedente è stato illustrato il ragionamento alla base degli alberi decisionali ed emergono dalla loro varietà di utilizzo sia dei vantaggi che degli svantaggi. Nonostante esistano diversi algoritmi con prestazioni più soddisfacenti, gli algoritmi degli alberi decisionali risultano significativamente utili per attività come il data mining, il machine learning o il knowledge discovery.

Considerando l'applicazione frequente in diversi ambiti reali, è possibile trarre diverse conclusioni sugli alberi, in quanto:

- Risultano facili da interpretare, grazie alla logica booleana e alle rappresentazioni visive che li contraddistinguono. La loro natura gerarchica permette anche una semplice visione di quali attributi sono più importanti;

- Non richiedono una precedente preparazione dei dati, difatti possono gestire vari tipi di dati discreti o continui, ad esempio variabili con valori mancanti, e possono convertire i valori continui in valori categoriali tramite l'uso di soglie.
- Sono flessibili, perché possono essere impiegati sia in attività di classificazione sia in attività di regressione. Inoltre, essendo insensibili alle relazioni tra gli attributi, quando due variabili sono altamente correlate, l'algoritmo ne sceglie soltanto una per effettuare la suddivisione.

Tuttavia, ci sono delle sfide da affrontare nell'utilizzo degli alberi, siccome risultano essere:

- Inclini all'overfitting, caratteristica spesso frequente negli alberi decisionali più complessi. Questo problema può essere evitato attraverso le attività di pre-potatura e post-potatura. Mediante la pre-potatura, è possibile interrompere la crescita nei casi in cui ci sono dati insufficienti, mentre la post-potatura si effettua dopo la creazione dell'albero, eliminando gli alberi secondari contenenti dati inadeguati;
- Stimatori con varianza elevata, infatti piccole variazioni dei dati possono produrre alberi molto differenti. Tale criticità è risolvibile con il metodo *bagging*, o media delle stime, il quale permette di ridurre notevolmente la varianza degli alberi decisionali, però va utilizzato con moderazione poiché può portare a predittori con elevata correlazione;
- Costosi, in quanto durante l'attività di creazione gli alberi adottano un approccio di ricerca che è più dispendioso rispetto ad altri algoritmi disponibili.

Per sintetizzare, è possibile osservare dunque che il vantaggio principale dell'uso degli alberi decisionali è che sono molto intuitivi e facili da spiegare. Rispetto agli altri metodi di regressione e classificazione, rispecchiano fedelmente il processo decisionale umano. Inoltre, possono essere visualizzati graficamente e possono gestire fattori predittivi qualitativi senza la necessità di creare variabili fittizie.

Ad ogni modo, il livello di accuratezza predittiva degli alberi decisionali è inferiore rispetto ad altri approcci, a causa della loro scarsa robustezza: anche una piccola modifica dei dati, come già anticipato, può causare un'ampia variazione nella stima finale dell'albero.

Tuttavia, aggregando molti alberi decisionali, grazie all'utilizzo di metodi come *random forest*, *bagging*, e *boosting*, le loro prestazioni predittive possono essere notevolmente migliorate.

1.3 Bagging, Random Forest, Boosting

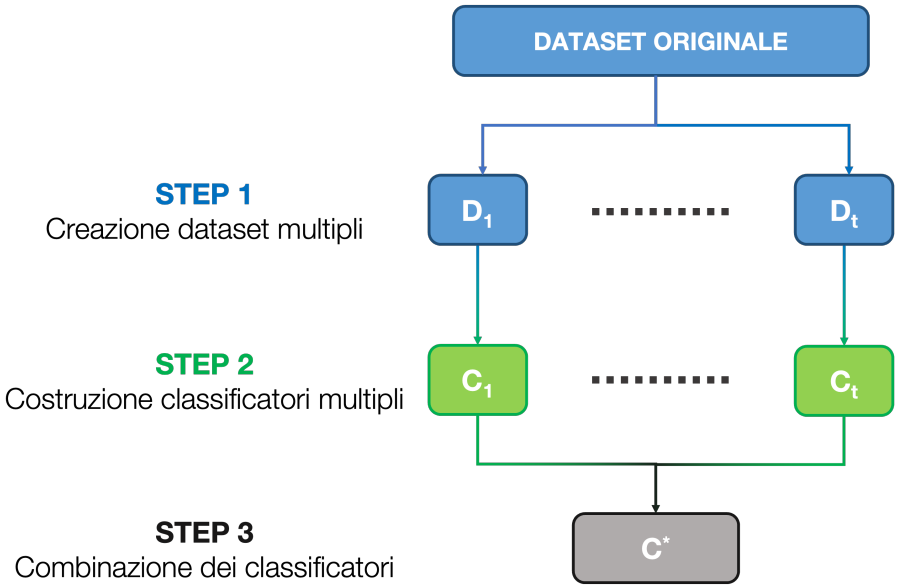
Per il miglioramento dell'accuratezza predittiva, è possibile combinare più alberi per ottenere una previsione aggregata. *bagging*, *random forests* e *boosting* sono alcuni approcci che implementano tale strategia e nei paragrafi successivi verrà dettagliato il loro funzionamento.

1.3.1 Bagging

La prima metodologia, chiamata *bagging* e nota anche come aggregazione del bootstrap, viene impiegata per ridurre la varianza di un dataset in 3 passaggi paralleli a partire dai dati originali del training set:

1. **Creare dataset multipli**: in questo step vengono rimpiazzati i dati originali con i nuovi dati tipicamente formati da una porzione delle righe e delle colonne dei dati iniziali. Questi sottoinsiemi possono essere usati come iper-parametri nel modello di *bagging*;

2. **Costruire classificatori multipli**: per ogni training set D_i si crea un classificatore C_i univoco;
3. **Combinare i classificatori**: per combinare le previsioni di tutti i classificatori individuati si utilizza il valore medio o una maggioranza delle previsioni per calcolare una stima più accurata, a seconda del problema da trattare.



Applicando questa tecnica agli alberi di regressione e di classificazione, si costruiscono semplicemente t alberi utilizzando t training set. Nel caso della regressione, viene calcolata la media delle previsioni risultanti (votazione soft), mentre per la classificazione, viene accettata la classe con la maggioranza dei voti (votazione hard). Si lasciano, inoltre, crescere gli alberi senza applicare l'attività di potatura.

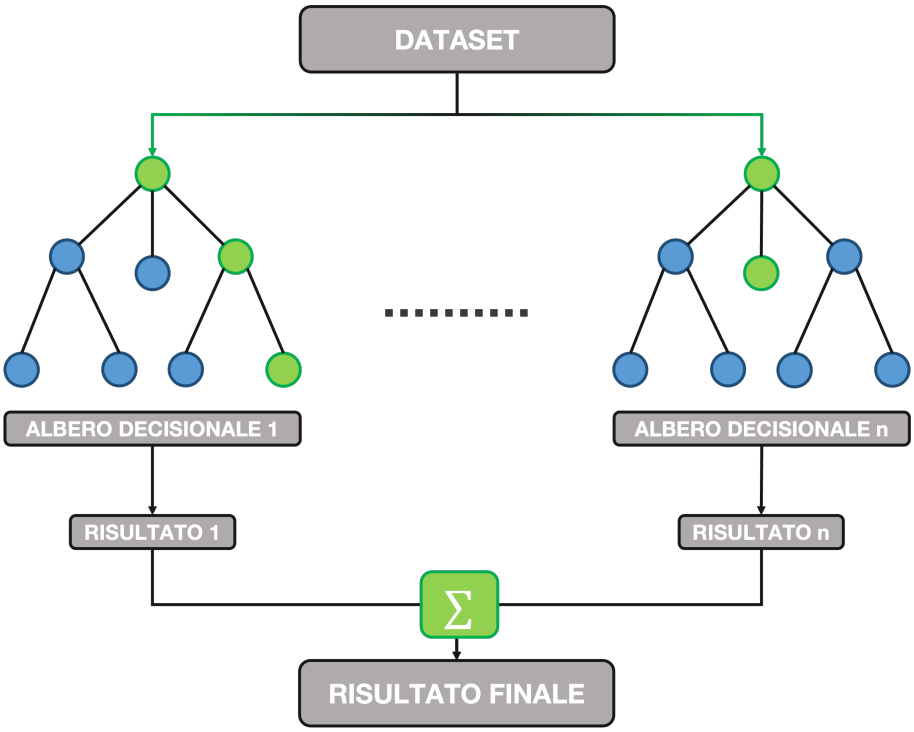
Esiste una serie di vantaggi e svantaggi che il bagging presenta nei problemi basati sugli alberi decisionali:

Vantaggi	Svantaggi
Facile interpretazione	Interpretabilità ridotta
Varianza ridotta	Computazione lenta
	Flessibilità ridotta

Sebbene la riduzione della varianza negli algoritmi di classificazione e regressione torni utile, il prezzo da pagare per una maggiore accuratezza è una perdita di interpretabilità e un rallentamento a livello computazionale, causati dall'aumento del numero di iterazioni che non rendono adeguata questa tecnica per applicazioni in tempo reale. Inoltre, una riduzione della flessibilità comporta che il bagging funzioni meglio con gli algoritmi poco stabili composti da un elevato numero di dati, dal momento che presentano una maggiore variazione nel dataset.

1.3.2 Random Forest

Il secondo approccio è il *random forest*, un algoritmo che combina l'output di più alberi decisionali per ottenere un unico risultato. Rispetto al *bagging*, questa tecnica offre un miglioramento in quanto elimina la correlazione tra gli alberi analizzati.



Di fatto, il *random forest* è soltanto una estensione del *bagging*, in quanto utilizza sia il *bagging* che la casualità per formare una "foresta" di alberi decisionali non correlati. L'algoritmo contiene tre iper-parametri principali che devono essere impostati prima della scelta del training set ed includono la dimensione dei nodi, il numero di strutture ad albero ed il numero di caratteristiche campionate. Il training set viene diviso inizialmente in due parti, la prima parte è identificata come test set, noto anche come campione *out-of-bag* usato per finalizzare la previsione tramite cross-validation, mentre la seconda parte viene inserita tramite la tecnica del *bagging* in modo casuale, il che permette di assegnare maggiore diversità al dataset e di ridurre la correlazione tra gli alberi. Successivamente, il classificatore del *random forest* viene impiegato per risolvere problemi di regressione e di classificazione. Anche in questo caso, per un'attività di regressione, verrà calcolata la media dei singoli alberi e per un'attività di classificazione, verrà considerato il voto di maggioranza, per esempio la variabile categoriale che si presenta più frequentemente, il quale produrrà la classe prevista.

Anche per questo approccio, sono stati individuati vantaggi e svantaggi che ne derivano dal suo utilizzo:

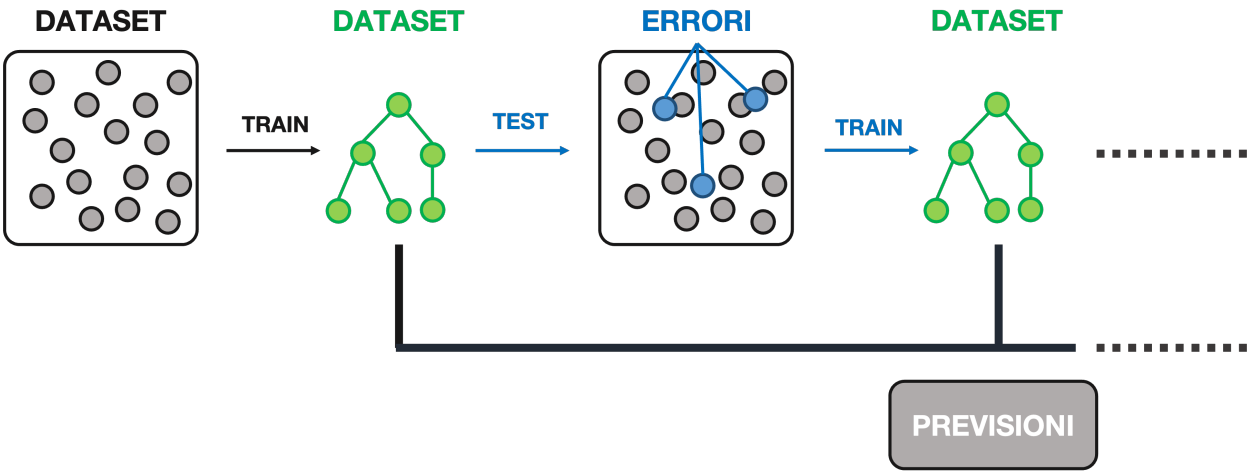
Vantaggi	Svantaggi
Minore rischio di overfitting	Necessità di un numero elevato di risorse
Flessibilità	Computazione lenta
Facile valutazione delle variabili	Complessità elevata

L'utilizzo del metodo *random forest* fornisce maggiore flessibilità perché può gestire sia la regressione che la classificazione con un alto grado di precisione, in aggiunta è uno strumento molto efficace per la stima di valori mancanti.

Inoltre, prevede una riduzione importante del rischio di overfitting, permettendo di valutare più facilmente il contributo delle variabili per il modello. Tuttavia, le sfide di tale approccio si riconoscono nella complessità di interpretazione, nella lentezza di elaborazione dei dati e nell'elevata necessità di risorse per la memorizzazione dei dati.

1.3.3 Boosting

Il terzo metodo è il *boosting*, utilizzato per il miglioramento dell'accuratezza predittiva degli alberi decisionali. Il funzionamento è simile al *bagging*, ad esclusione del fatto che gli alberi vengono fatti crescere in sequenza, utilizzando le informazioni degli alberi costruiti in precedenza, compensando i punti deboli dei predecessori. A differenza del *bagging* però, il *boosting* non coinvolge il bootstrap, perché ogni albero viene adattato ad una versione modificata del data set iniziale. Ad ogni iterazione, viene migliorata l'accuratezza predittiva, ricombinando i classificatori precedenti.



Di seguito vengono riportati i principali vantaggi e svantaggi dell'approccio:

Vantaggi	Svantaggi
Facile implementazione	Incline all'overfitting
Computazione efficiente	Computazione costosa
Bias ridotto	

Il *boosting* si rivela molto utile quando si dispone di un elevato numero di dati e si prevede che gli alberi decisionali siano molto complessi. L'utilizzo è indicato principalmente per risolvere i problemi di classificazione e regressione, tra cui l'analisi del rischio, la sentiment analysis, la modellazione dei prezzi, la stima delle vendite e la diagnosi dei pazienti.

2 Utilizzo e confronto pacchetti

Per utilizzare un pacchetto, prima di tutto è essenziale ottenerlo. Nel caso questo faccia parte della repository CRAN, questo può essere fatto attraverso il comando *install.packages()* al quale viene passato il nome del pacchetto.

```
install.packages("PackageName")
```

A volte può essere necessario dover installare un pacchetto specifico non da CRAN ma da un'altra repository, ad esempio da GitHub. In questi casi, è necessario seguire i seguenti passaggi:

```
install.packages("devtools")
devtools::install_github("GithubRepositoryPat")
```

Nel caso in cui il pacchetto non si trovasse né su CRAN né su GitHub, ma su un altro repository esterno, è necessario usare nuovamente la funzione *install.packages()* con due parametri opzionali, *repos*, che rappresenta l'URL del repository e *dependencies*, che rappresenta se le dipendenze verranno installate o meno.

```
install.packages("PackageName", repos="URL", dependencies=TRUE)
```

Oltre a queste metodologie, il processo di installazione può essere svolto anche attraverso l'interfaccia di Rstudio.

Nel caso in cui fosse necessario aiuto oppure maggiori informazioni riguardo ai pacchetti, è possibile sfruttare la sezione *Help* di R. Questa, infatti, include ampie possibilità di accesso alla documentazione e di ricerca aiuto, come ad esempio la funzione *help()* e l'operatore *?* che consentono di accedere alle pagine di documentazione delle funzioni, dei set di dati e di altri oggetti di R.

```
?tree
help(rpart)
```

Dopo aver installato i pacchetti, il passo successivo è caricarli. Questo è necessario per ogni nuova sessione di R, a differenza dell'installazione. A questo scopo, si utilizza la funzione *library()* alla quale si passa il nome del pacchetto.

```
library(PackageName)
```

2.1 Il pacchetto tree

Il pacchetto R *tree*, attualmente in versione 1.0-43, è un pacchetto svillupato e aggiornato da Brian Ripley per lavorare con gli alberi decisionali. Questo permette di sviluppare, modificare ed elaborare gli alberi di classificazione e regressione nella programmazione R.

```
library(tree)
```

2.1.1 Funzioni principali

Di seguito verranno descritte le funzioni principali del pacchetto *tree*, approfondendone l'utilizzo e gli argomenti che richiedono.

tree()

Permette la costruzione di un albero decisionale. Il valore ottenuto è un oggetto *tree*, con i seguenti componenti:

- *frame:* un dataframe con una riga per ogni nodo. Le colonne includono:
  - *row.names:* fornisce i numeri dei nodi con ordinamento binario indicizzato dalla profondità del nodo;
  - *var:* la variabile utilizzata nello split (o "foglia" per un nodo terminale);
  - *n:* il numero (ponderato) di casi che raggiungono quel nodo;
  - *dev:* la devianza del nodo;
  - *yval:* il valore adattato al nodo (la media per gli alberi di regressione, una classe di maggioranza per gli alberi di classificazione);
  - *split:* matrice a due colonne delle etichette per gli split a sinistra e a destra del nodo;
  - *yprob:* unicamente per gli alberi di classificazione, una matrice di probabilità adattata per ogni livello di risposta.
- *where:* un vettore di interi che indica il numero di riga del frame che specifica il nodo a cui è assegnato ogni caso;
- *terms:* i termini della *formula*;
- *call:* la chiamata a *tree*;
- *model:* modello del *frame*;
- *x:* matrice del modello;
- *y:* variabile di risposta;
- *wts:* pesi.

L'albero viene costruito mediante partizione binaria ricorsiva utilizzando la variabile di risposta nella formula e scegliendo gli split dalle variabili esplicative. Le variabili numeriche sono suddivise in $X > a$ e $X < a$. Si sceglie split che massimizzano la riduzione dell'impurità, dividendo il set di dati e ripetendo il processo. La suddivisione continua finché i nodi terminali non sono troppo piccoli o troppo pochi per essere suddivisi. La crescita dell'albero è limitata a una profondità di 31 livelli, utilizzando numeri interi per etichettare i nodi.

Esempio di utilizzo della funzione:

```
tree(formula, data, weights, subset,
     na.action = na.pass, control = tree.control(nobs, ...),
     method = "recursive.partition",
     split = c("deviance", "gini"),
     model = FALSE, x = FALSE, y = TRUE, wts = TRUE, ...)
```

Parametri:

- *formula:* un'espressione nella quale il lato sinistro (risposta) deve essere un vettore numerico, se viene applicato un albero di regressione, o un fattore, se viene prodotto un albero di classificazione. Il lato destro deve essere composto da una serie di variabili numeriche o fattoriali separate da +, senza termini di interazione (il . è ammesso);
- *data:* un dataframe per l'interpretazione di *formula*, *weights* e *subset*;
- *weights:* vettore di pesi non negativi; sono ammessi pesi frazionari;
- *subset:* un'espressione che specifica il sottoinsieme di osservazioni da utilizzare;
- *na.action:* funzione per filtrare i dati mancanti;
- *control:* una lista restituita da *tree.control*;
- *method:* stringa indicante il metodo da utilizzare (*recursive.partition* oppure *model.frame*);
- *split:* criterio di suddivisione da utilizzare;
- *model:* se questo argomento è un *model frame*, gli argomenti *formula* e *data* vengono ignorati e viene usato per definire il modello. Se *"model=TRUE"*, il modello viene memorizzato come componente *model* nel risultato;
- *x:* se vero, viene restituita la matrice delle variabili per ogni osservazione;
- *y:* se vero, viene restituita la variabile di risposta;
- *wts:* se vero, vengono restituiti i pesi;

tree.control()

Utile funzione per il controllo dei parametri di costruzione dell'albero. Restituisce un vettore composto da:

- *mincut:* il massimo tra il valore di *mincut* e 1;
- *minsize:* il massimo del valore di *minsize* e 2;
- *nmax:* stima del numero massimo di nodi che potrebbero crescere;
- *nobs:* numero di osservazioni in input.

Inoltre, produce valori predefiniti di *mincut* e *minsize* e assicura che *mincut* sia al massimo la metà di *minsize.*

Esempio di utilizzo della funzione:

```
tree.control(nobs, mincut = 5, minsize = 10, mindev = 0.01)
```

Parametri:

- *nobs:* numero di osservazioni nel training set;
- *mincut:* il numero minimo di osservazioni da includere in uno dei due nodi figli (default = 5);
- *minsize:* dimensione minima consentita del nodo (default = 10);

- *mindev:* la devianza interna del nodo deve essere almeno quella del nodo radice moltiplicata per questo coefficiente affinchè il nodo venga diviso.

deviance.tree()

Necessario per estrarre la devianza da un oggetto *tree*. Il valore sarà pari alla devianza complessiva o un vettore di contributi derivanti dai nodi (la devianza complessiva è la somma delle devianze delle foglie).

Esempio di utilizzo della funzione:

```
deviance(object, detail = FALSE, ...)
```

Parametri:

- *object:* un oggetto *tree*;
- *detail:* se vero, restituisce un vettore dei contributi di devianza di ciascun nodo;
- *. . . :* argomenti da passare a/da altri metodi.

cv.tree()

Esegue una K-fold cross-validation per trovare la devianza o il numero di classificazioni errate in funzione del parametro di costo-complessità K.

Esempio di utilizzo della funzione:

```
cv.tree(object, rand, FUN = prune.tree, K = 10, ...)
```

Parametri:

- *object:* un oggetto *tree*;
- *rand:* parametro opzionale. Vettore di interi casuali della lunghezza del numero di casi utilizzati per creare l'oggetto *tree*, che assegna i casi a folds diverse per la cross-validation;
- *FUN:* funzione che esegue la potatura;
- *K:* il numero di folds della cross-validation;
- *. . . :* argomenti aggiuntivi di *FUN*.

na.tree.replace()

Questa funzione viene utilizzata tramite il parametro *na.action* di **tree()**. Aggiunge un nuovo livello chiamato "NA" a qualsiasi *predittore discreto* in un dataframe che contiene NA. Si ferma se qualsiasi *predittore continuo* contiene una NA.

Esempio di utilizzo della funzione:

```
na.tree.replace(frame)
```

Parametri:

- *frame:* dataframe utilizzato per creare un albero.

predict.tree()

Restituisce un vettore di previsioni di risposta da un *fitted tree object*. Questa funzione è un metodo per la classe *tree* della funzione generica *predict()*. Può essere invocata chiamando *predict(x)* per un oggetto x della classe opportuna, oppure direttamente chiamando *predict.tree(x)* indipendentemente dalla classe dell'oggetto.

Esempio di utilizzo della funzione:

```
predict(object, newdata = list(),
        type = c("vector", "tree", "class", "where"),
        split = FALSE, nwts, eps = 1e-3, ...)
```

Parametri:

- *object:* un *fitted tree object*. Si assume che questo sia il risultato di una funzione che produce un oggetto con gli stessi componenti di quello restituito dalla funzione **tree()**;
- *newdata:* dataframe contenente i valori per i quali sono richieste le previsioni. I predittori a cui si fa riferimento nella parte destra di *formula* devono essere presenti in *newdata*. Se mancano, vengono restituiti i *fitted values*;
- *type:* stringa indicante l'output (default = "vector"):
  - *type = "vector":* vettore di previsioni di risposta o, se la risposta è un fattore, matrice di probabilità di classi previste;
  - *type = "tree":* restituisce un oggetto di classe *tree* con nuovi valori per le variabili *n* e *dev* di *frame*;
  - *type = "class":* per un albero di classificazione, un fattore delle classi previste;
  - *type = "where":* i nodi raggiunti dai casi.
- *split:* regola la gestione dei valori mancanti:
  - *split=FALSE*, i casi con valori mancanti vengono fatti scendere lungo l'albero fino a raggiungere una foglia o un nodo per il quale l'attributo è mancante, e questo nodo viene usato per la previsione;
  - *split=TRUE*, i casi con attributi mancanti vengono suddivisi in frazioni di casi e abbandonati su ciascun lato dello split. I valori previsti vengono calcolati come media delle frazioni per ottenere la previsione.
- *nwts:* pesi per i casi dei nuovi dati, usati quando si prevede un albero;
- *eps:* limite inferiore per le probabilità, utilizzato se nei nuovi dati si verificano eventi con probabilità zero durante la previsione;
- *. . . :* argomenti da passare a/da altri metodi.

misclass.tree()

Riporta il numero di classificazioni errate effettuate da un albero di classificazione, sia nel complesso che in ciascun nodo (le quantità restituite sono ponderate con i pesi di osservazione, se forniti).

Esempio di utilizzo della funzione:

```
misclass.tree(tree, detail = FALSE)
```

Parametri:

- *tree:* un oggetto *tree*;
- *detail:* se falso, riporta il numero complessivo di classificazioni errate. Se vero, riporta il numero per ogni nodo.

prune.tree() e prune.misclass()

Determina una sequenza annidata di sottoalberi dell'albero fornito, "tagliando" in modo ricorsivo gli split meno importanti in base alla misura di costo-complessità. *prune.misclass* è un'abbreviazione di *prune.tree(method = "misclass")*.

Esempio di utilizzo delle funzioni:

```
prune.tree(tree, k = NULL, best = NULL, newdata, nwts,
           method = c("deviance", "misclass"), loss, eps = 1e-3)

prune.misclass(tree, k = NULL, best = NULL, newdata,
               nwts, loss, eps = 1e-3)
```

Parametri:

- *tree:* un *fitted tree object*. Si assume che questo sia il risultato di una funzione che produce un oggetto con gli stessi componenti di quello restituito dalla funzione *tree()*;
- *k:* parametro di costo-complessità che definisce uno specifico sottoalbero dell'albero (*k* uno scalare) o sequenza di sottoalberi che minimizzano la misura di costo-complessità (*k* un vettore). Se manca, viene determinato algoritmicamente;
- *best:* intero che richiede la dimensione (numero di nodi terminali) di uno specifico sottoalbero nella sequenza di costo-complessità da restituire. Si tratta di un modo alternativo di selezionare un sottoalbero rispetto a *k*. Se non esiste un albero della dimensione richiesta, viene restituito il successivo più grande;
- *newdata:* dataframe su cui viene valutata la misura di costo-complessità. Se manca, vengono utilizzati i dati usati per far crescere l'albero;
- *nwts:* pesi per i casi dei nuovi dati;
- *method:* stringa indicante la misura dell'eterogeneità dei nodi utilizzata per guidare la potatura. Per gli alberi di regressione, è accettato solo il valore predefinito, la *devianza*. Per gli alberi di classificazione, il valore predefinito è la *devianza* e l'alternativa è *misclass* (numero di classificazioni errate o perdita totale);

- *loss:* matrice che fornisce per ogni classe vera (riga) la perdita numerica della previsione della classe (colonna). Le classi devono essere nell'ordine dei livelli della risposta;
- *eps:* limite inferiore per le probabilità, utilizzato per calcolare le deviazioni se nei nuovi dati si verificano eventi con probabilità zero.

Se *k* o *best* non vengono forniti, viene restituito un oggetto di classe *tree.sequence*, contenente i seguenti componenti:

- *size:* numero di nodi terminali in ogni albero della sequenza di potatura;
- *deviance:* devianza totale di ogni albero nella sequenza di potatura;
- *k:* valore del parametro di potatura.

2.1.2 Altre funzioni

Di seguito vengono mostrate altre funzioni secondarie del pacchetto *tree*, senza approfondirne l'utilizzo e gli argomenti che richiedono.

Funzione	Descrizione
`plot.tree()`	Stampa di un oggetto *tree*
`plot.tree.sequence()`	Stampa di una sequenza di oggetto *tree*
`text.tree()`	Aggiunta testo a un grafico ad albero
`snip.tree()`	Se "nodes" viene indicato, rimuove il nodo e tutti i suoi discendenti dall'albero Se non viene indicato "nodes" , è necessario selezionare manualmente il nodo per la stampa del suo numero e della divergenza tra l'albero attuale e quello che rimarrebbe se quel nodo venisse rimosso. Selezionandolo nuovamente viene rimosso
`partition.tree()`	Stampa le partizioni di un albero che coinvolge una o due variabili
`tile.tree()`	Calcolo delle frequenze di livello di variabili per i casi che raggiungono ogni foglia dell'albero e stampa dei grafici a barre dell'insieme di frequenze sotto ogni foglia
`tree.screens()`	Divisione dello schermo per uso di *tile.tree()*

2.2 Il pacchetto rpart

Il pacchetto R *rpart*, attualmente in versione 4.1.19, è un pacchetto svillupato da Terry Therneau, Beth Atkinson e Brian Ripley. Questo pacchetto costruisce modelli di classificazione o regressione utilizzando una procedura a due stadi, ottenendo modelli che possono essere rappresentati come alberi binari. Il pacchetto implementa molte delle idee contenute nel libro e nei programmi **CART (Classification and Regression Trees)** di Breiman, Friedman, Olshen e Stone. Il nome del pacchetto deriva da Recursive PARTitioning o *rpart*.

```
library(rpart)
```

2.2.1 Funzioni principali

Di seguito verranno descritte le funzioni principali del pacchetto *rpart*, approfondendone l'utilizzo e gli argomenti che richiedono e confrontandole con il pacchetto *tree* visto in precedenza.

rpart()

L'output è un oggetto *rpart*. Si differenzia dalla funzione *tree* soprattutto per la gestione delle variabili surrogate.

I seguenti componenti sono inclusi in un oggetto *rpart*:

- *frame:* un dataframe con una riga per ogni nodo. Le colonne includono:
  - *row.names:* fornisce i numeri dei nodi con ordinamento binario indicizzato dalla profondità del nodo;
  - *var:* fornisce i nomi delle variabili utilizzate negli split in ogni nodo (nodi foglia indicati con livello "leaf");
  - *n:* numero di osservazioni che raggiungono il nodo;
  - *wt:* somma dei pesi dei casi per le osservazioni che raggiungono il nodo;
  - *dev:* devianza del nodo;
  - *yval:* valore adattato della risposta al nodo;
  - *split:* matrice a due colonne di etichette di split a sinistra e a destra per ogni nodo;
  - *complexity:* parametro di complessità al quale lo split collassa;
  - *ncompete:* numero di split concorrenti registrati;
  - *nsurrogate:* numero di split surrogati registrati.
- *where:* vettore di interi di lunghezza pari al numero di osservazioni nel nodo radice, contenente il numero di riga del frame corrispondente al nodo foglia in cui rientra ogni osservazione;
- *call:* chiamata che ha prodotto l'oggetto;
- *terms:* oggetto, utilizzato da vari metodi, di classe *c("terms", "formula")* che riassume la formula;
- *splits:* matrice numerica che descrive gli split;
- *csplit:* matrice di interi, formata da una riga per ogni split mentre il numero di colonne è il numero maggiore di livelli nei fattori;
- *method:* metodo utilizzato per far crescere l'albero (*"class", "exp", "poisson", "anova"* o *"user"*);
- *cptable:* matrice di informazioni sulle potature ottimali basate su un parametro di complessità;
- *variable.importance:* vettore numerico che fornisce l'importanza di ogni variabile. Quando vengono stampati da *summary.rpart*, vengono ridimensionati in modo che la somma sia pari a 100;
- *numresp:* numero di livelli di risposta di un fattore;
- *parms, control:* record degli argomenti forniti;
- *functions:* le funzioni *summary*, *print* e *text* per il metodo utilizzato;
- *ordered:* vettore che registra per ogni variabile se si tratta di un fattore ordinato;

- *na.action:* informazioni sulla gestione dei NA.

Esempio di utilizzo della funzione:

```
rpart(formula, data, weights, subset, na.action = na.rpart, method,
      model = FALSE, x = FALSE, y = TRUE, parms, control, cost, ...)
```

Parametri:

- *formula:* un'espressione nella quale il lato sinistro (risposta) deve essere un vettore numerico, se viene applicato un albero di regressione, o un fattore, se viene prodotto un albero di classificazione. Il lato destro deve essere composto da una serie di variabili numeriche o fattoriali separate da +, senza termini di interazione (il . è ammesso);
- *data:* dataframe opzionale con cui interpretare le variabili nominate nella formula;
- *weights:* pesi opzionali per i casi;
- *subset:* opzionale, indica che solo un sottoinsieme delle osservazioni deve essere utilizzato nel fit;
- *na.action:* di default elimina tutte le osservazioni per le quali y (response) è mancante, mantenendo quelle in cui uno o più predittori sono mancanti;
- *method:* metodo tra *"anova"*, *"poisson"*, *"class"* o *"exp"*. Se il metodo è mancante, viene effettuata un'ipotesi autonoma;
- *model:* se questo argomento è un *model frame*, gli argomenti *formula* e *data* vengono ignorati e viene usato per definire il modello. Se *"model=TRUE"*, il modello viene memorizzato come componente *model* nel risultato;
- *x:* se vero, mantiene una copia della matrice x nel risultato;
- *y:* se vero, mantiene una copia della variabile dipendente nel risultato (*default=FALSE*);
- *parms:* parametri opzionali per la funzione di splitting:
  - *Anova splitting* non ha parametri;
  - *Poisson splitting* ha un solo parametro, il coefficiente di variazione della distribuzione precedente (*default=1*);
  - *Exponential splitting* ha lo stesso parametro di Poisson;
  - *Classification splitting* l'elenco può contenere: il vettore delle probabilità antecedenti (*prior component*), la matrice di perdita (*loss component*) o l'indice di splitting (*split component*).
- *control:* elenco di opzioni che controllano i dettagli dell'algoritmo *rpart.control*;
- *cost:* vettore di costi non negativi, uno per ogni variabile del modello. Il valore predefinito è uno per tutte le variabili. Nel decidere quale suddivisione scegliere, il miglioramento dello splitting di una variabile viene diviso per il suo costo;
- *. . . :* gli argomenti di *rpart.control* possono anche essere specificati nella chiamata a rpart.

rpart.control()

Utilizzata per gestire diversi parametri che controllano il fit di *rpart()*.

Esempio di utilizzo della funzione:

```
rpart.control(minsplit = 20, minbucket = round(minsplit/3), cp = 0.01,
              maxcompete = 4, maxsurrogate = 5, usesurrogate = 2, xval = 10,
              surrogatestyle = 0, maxdepth = 30, ...)
```

Parametri:

- *minsplit:* numero minimo di osservazioni in un nodo per poter effettuare uno split;
- *minbucket:* numero minimo di osservazioni in qualsiasi nodo terminale (nodo foglia). Se viene specificato solo uno dei due parametri *minbucket* o *minsplit*, il codice imposta $minsplit = minbucket * 3$;
- *cp:* parametro di complessità. Qualsiasi split che non riduca l'inadeguatezza complessiva di un fattore pari a cp non viene tentato;
- *maxcompete:* numero di split "rivali" mantenuti nell'output;
- *maxsurrogate:* numero di split surrogati mantenuti nell'output. Se questo valore è impostato a zero, il tempo di calcolo sarà ridotto, poiché circa la metà del tempo di calcolo viene utilizzato nella ricerca delle suddivisioni surrogate;
- *usesurrogate:* determina come utilizzare i surrogati nel processo di suddivisione:
  - *usesurrogate=0:* indica unicamente visualizzazione, ovvero un'osservazione con valore mancante per la regola di divisione primaria non viene inviata più avanti nell'albero;
  - *usesurrogate=1:* indica l'utilizzo dei surrogati. Se tutti i surrogati mancano l'osservazione non viene divisa;
  - *usesurrogate=2:* indica l'utilizzo dei surrogati. Se tutti i surrogati mancano, l'osservazione viene inviata nella direzione della maggioranza.
- *xval:* numero di folds della cross-validation (può anche essere un vettore di interi);
- *surrogatestyle:* controlla la selezione del miglior surrogato. Se impostato su 0 (default) utilizza il numero totale di classificazioni corrette per una potenziale variabile surrogata, se impostato su 1 utilizza la percentuale corretta, calcolata sui valori non mancanti del surrogato;
- *maxdepth:* imposta la profondità massima di qualsiasi nodo dell'albero finale, con il nodo radice contato come profondità 0;
- *. . . :* altri argomenti.

na.rpart()

Gestisce i valori mancanti in un oggetto *rpart*. Omette le osservazioni in cui parte della risposta è mancante o tutte le variabili esplicative sono mancanti.

Esempio di utilizzo della funzione:

```
na.rpart(x)
```

Parametri:

- *x: model frame*.

plotcp() e printcp()

plotcp() fornisce una rappresentazione visiva dei risultati della cross-valiation di un oggetto *rpart*.

printcp() visualizza la tabella cp per un fitted *rpart* object e stampa una tabella di potature ottimali basate su un parametro di complessità.

Esempio di utilizzo delle funzioni:

```
plotcp(x, minline = TRUE, lty = 3, col = 1,
       upper = c("size", "splits", "none"), ...)

printcp(x, digits = getOption("digits") - 2)
```

Parametri *plotcp()*:

- *x:* un *fitted rpart object*, risultato di una funzione *rpart*;
- *minline:* se vero, viene tracciata una linea orizzontale sopra il minimo della curva;
- *lty:* tipo linea;
- *col:* colore linea;
- *upper:* ciò che viene stampato sull'asse superiore tra dimensione dell'albero (numero di foglie), numero di divisioni o nulla;
- *. . . :* parametri di stampa aggiuntivi.

Parametri *printcp()*:

- *x:* un *fitted rpart object*, risultato di una funzione *rpart*;
- *digits:* numero di cifre.

predict.rpart()

Restituisce un vettore di previsioni per la variabile di risposta da un *fitted rpart object*. Questa funzione è un metodo per la classe *rpart* della funzione generica *predict()*. Può essere invocata chiamando *predict(x)* per un oggetto x della classe opportuna, oppure direttamente chiamando *predict.rpart(x)* indipendentemente dalla classe dell'oggetto.

Esempio di utilizzo della funzione:

```
predict(object, newdata,
       type = c("vector", "prob", "class", "matrix"),
       na.action = na.pass, ...)
```

Parametri:

- *object:* un *fitted model object* di classe *rpart*, risultato del metodo *rpart*;
- *newdata:* dataframe contenente i valori per i quali sono richieste le previsioni. I predittori a cui si fa riferimento nella parte destra di *formula* devono essere presenti in *newdata*. Se mancano, vengono restituiti i *fitted values*;

- *type:* stringa indicante il tipo di valore predetto restituito. Se l'oggetto *rpart* è un albero di classificazione, l'impostazione predefinita è quella di restituire *prob* (corrispondente al comportamento predefinito di tree). Altre tipologie:
  - *type="vector":* vettore di previsioni della variabile di risposta. Per alberi di regressione è la risposta media al nodo e per alberi di classificazione è la classe prevista;
  - *type="prob":* per alberi di classificazione, una matrice di probabilità delle classi;
  - *type="matrix":* matrice delle risposte. Per alberi di regressione è la risposta media, per alberi di classificazione è la concatenazione di almeno la classe prevista, il numero di classi in quel nodo dell'albero adattato e le probabilità di classe;
  - *type="class":* per alberi di classificazione, un fattore di classificazione basato sulle risposte.
- *na.action:* funzione che determina cosa fare con i valori mancanti nei nuovi dati (*na.pass*, *na.omit* o *na.fail*);
- *. . . :* argomenti da passare a/da altri metodi.

prune.rpart()

Determina una sequenza di sottoalberi dell'oggetto *rpart* fornito, "tagliando" ricorsivamente le suddivisioni meno importanti, in base al parametro di complessità *cp*, restituendo un nuovo oggetto *rpart* "tagliato" al valore *cp*.

Esempio di utilizzo della funzione:

```
prune(tree, cp, ...)
```

Parametri:

- *tree:* un *fitted model object* di classe *rpart*, risultato di una funzione *rpart*;
- *cp:* parametro di complessità a cui l'oggetto *rpart* sarà potato;
- *. . . :* argomenti da passare a/da altri metodi.

residuals.rpart()

Permette di visualizzare i residui di un oggetto *rpart*.

Esempio di utilizzo della funzione:

```
residuals(object, type = c("usual", "pearson", "deviance"), ...)
```

Parametri:

- *object:* un *fitted model object* di classe *rpart*;
- *type:* indica il tipo di residuo desiderato;
- *. . . :* argomenti da passare a/da altri metodi.

xpred.rpart()

Fornisce i valori previsti per il fit di un oggetto *rpart*, attraverso cross-validation, per un insieme di valori dei parametri di complessità.

Esempio di utilizzo della funzione:

```
xpred.rpart(fit, xval = 10, cp, return.all = FALSE)
```

Parametri:

- *fit:* un oggetto di classe *rpart*;
- *xval:* numero di folds della cross-validation (può anche essere un vettore di interi);
- *cp:* lista dei valori di complessità. Di default, viene preso da cptable;
- *return.all:* se falso, restituisce solo il primo elemento della previsione.

2.2.2 Altre funzioni

Di seguito vengono mostrate altre funzioni secondarie del pacchetto *rpart*, senza approfondirne l'utilizzo e gli argomenti che richiedono.

Funzione	Descrizione
rsq.rpart()	Produce 2 grafici, il primo traccia il R^2 rispetto al numero di suddivisioni, il secondo l'errore della cross-validation rispetto al numero di split
snip.rpart()	Crea un oggetto `rpart` "tagliato", contenente i nodi rimasti dopo che i sottoalberi selezionati sono stati potati
summary.rpart()	Restituisce informazioni dettagliate del fit di un oggetto `rpart`
meanvar.rpart()	Crea un grafico della devianza del nodo divisa per il numero di osservazioni sul nodo. Restituisce anche il numero del nodo
path.rpart()	Restituisce un elenco di nomi in cui ogni elemento contiene gli split sul percorso dalla radice ai nodi selezionati
plot.rpart()	Stampa il grafico di un oggetto `rpart`
print.rpart()	Stampa un oggetto `rpart`
labels.rpart()	Fornisce le labels per i rami di un albero
text.rpart()	Etichetta il grafico dell'albero con un testo
post.rpart()	Genera un grafico di presentazione PostScript di un oggetto

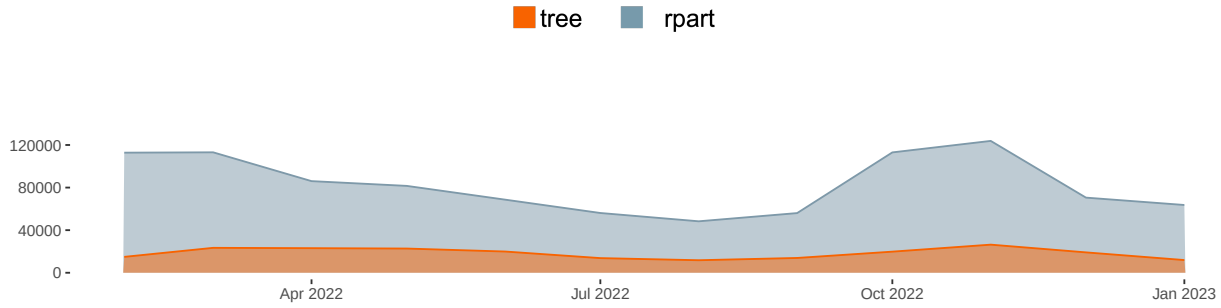
Oltre a queste, in *rpart* sono presenti funzioni per l'accesso diretto ad alcuni dataset; ad esempio:

Funzione	Descrizione
`solder.balance`	Dataset rappresentante un esperimento sulla saldatura di componenti su schede a circuito stampato
`stagec`	Dataset rappresentante pazienti con carcinoma prostatico in stadio C
`kyphosis`	Dataset rappresentante dati su bambini che hanno subito un intervento chirurgico correttivo alla colonna vertebrale
`car90`	Dataset relativo a vetture
`cu.summary`	Dataset rappresentante dati riguardo marche di automobili prodotte dal 1990
`car.test.frame`	Dataset estensione di **`cu.summary`**

2.3 Confronto

Entrambi i pacchetti analizzati risultano ottimi per la costruzione di alberi di classificazione e regressione. Ambedue, infatti, sono stati sviluppati sulla base del libro e degli algoritmi **CART** del 1984 di Breiman, Friedman, Olshen e Stone.

Secondo il sito web *RDocumentation* vi è una prevalenza nei download mensili del pacchetto *rpart* rispetto al pacchetto *tree* come mostrato nel grafico sottostante, indicante l'andamento dei download nell'ultimo anno.



La differenza principale tra i due è il modo in cui gestiscono i valori mancanti nei processi di splitting e valutazione. Con *tree*, un'osservazione con un valore mancante per la regola di divisione primaria non viene fatta scendere ulteriormente nell'albero. Con *rpart*, invece, gli utenti possono scegliere il modo in cui gestire i valori mancanti, compreso l'uso di surrogati, impostando il parametro *usesurrogate* nell'opzione *rpart.control*.

Sebbene entrambi i pacchetti offrano la possibilità di visualizzare i dettagli della suddivisione e di tracciare l'albero, con *rpart* questo è possibile in maniera più completa attraverso i comandi *summary(object)*, *plot(object)*, *plotcp(object)* e *print(object)*.

Oltre a questo, *rpart* offre una maggiore flessibilità nella crescita degli alberi, permettendo di impostare ben 9 parametri per il processo di modellazione, mentre *tree* permette l'inserimento di unicamente 3 parametri per il controllo del processo di modellazione.

3 Esempi applicativi

In questa sezione verranno mostrati due esempi di applicazione delle tecniche precedentemente analizzate, attraverso due dataset rispettivamente per gli alberi di classificazione e regressione.

3.1 Classificazione

Il dataset utilizzato per l'albero di classificazione è *clients* che raccoglie le informazioni di 1723 finanziamenti concessi ad altrettanti clienti, ed è composto da 14 colonne:

- *month*: mese in cui è stato richiesto il prestito (variabile quantitativa);
- *credit_amount*: ammontare del credito concesso (variabile quantitativa);
- *credit_term*: termine per la restituzione del credito (variabile quantitativa);
- *age*: età del richiedente (variabile quantitativa);
- *sex*: sesso del richiedente (variabile categoriale);
- *education*: livello di istruzione del richiedente (variabile categoriale);
- *product_type*: tipo di prodotto per cui è stato richiesto il finanziamento (variabile categoriale);
- *having_children_flg*: il richiedente ha figli o meno (variabile categoriale);
- *region*: regione di appartenenza del richiedente (variabile categoriale);
- *income*: reddito del richiedente (variabile quantitativa);
- *family_status*: stato di famiglia (variabile categoriale);
- *phone_operator*: operatore telefonico del richiedente (variabile categoriale);
- *is_client*: flag per clienti già presenti a sistema (variabile categoriale);
- *bad_client_target*: flag che identifica i cattivi pagatori (variabile dipendente categoriale).

L'obiettivo è sviluppare un modello che permetta di identificare correttamente i potenziali cattivi pagatori attibuendogli il flag *bad_client_target* in modo da non rischiare di concedere prestiti a dei clienti che in un futuro potrebbero non pagare i propri debiti.

È necessario dividere il dataset in due parti (training e testing) in modo da poter verificare in un secondo momento le prestazioni del modello sviluppato sulla base del subset di training, sfruttando il subset di testing.

3.1.1 Training

Nella fase di training, si crea l'albero impostando i parametri della funzione che si utilizzerà sulla base del dataset analizzato.

In questo caso è stata utilizzata la funzione *tree()*. I parametri che è possibile impostare per il controllo della crescita sono unicamente tre: *mincut*, *minsize* e *mindev*.

```
tree.model <- tree(bad_client_target ~ .,
                   data = Train.data,
                   control = tree.control(nrow(Train.data),
                                          mincut = 5,
                                          minsize = 10,
                                          mindev = 0.01))
```

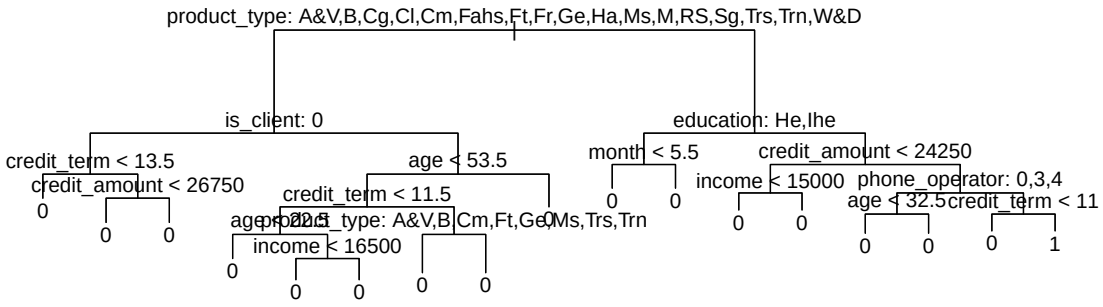
Lo stesso è stato fatto utilizzando la funzione *rpart()*. In questo caso i parametri di controllo sono nove: *minsplit*, *minbucket*, *cp*, *maxcompete*, *maxsurrogate*, *usesurrogate*, *xval*, *surrogatestyle* e *maxdepth*. Questa è una grossa differenza rispetto alla funzione *tree* utilizzata in precedenza che ci permette di gestire al meglio alberi di complessità elevata, garantendo un migliore fit.

```
rpart.model <- rpart(bad_client_target ~ .,
                     data = Train.data,
                     control = rpart.control(minbucket = round(20/3),
                                             cp = 0.0001,
                                             maxcompete = 6,
                                             maxsurrogate = 6,
                                             usesurrogate = 1,
                                             xval = 100,
                                             surrogatestyle = 1,
                                             maxdepth = 20))
```

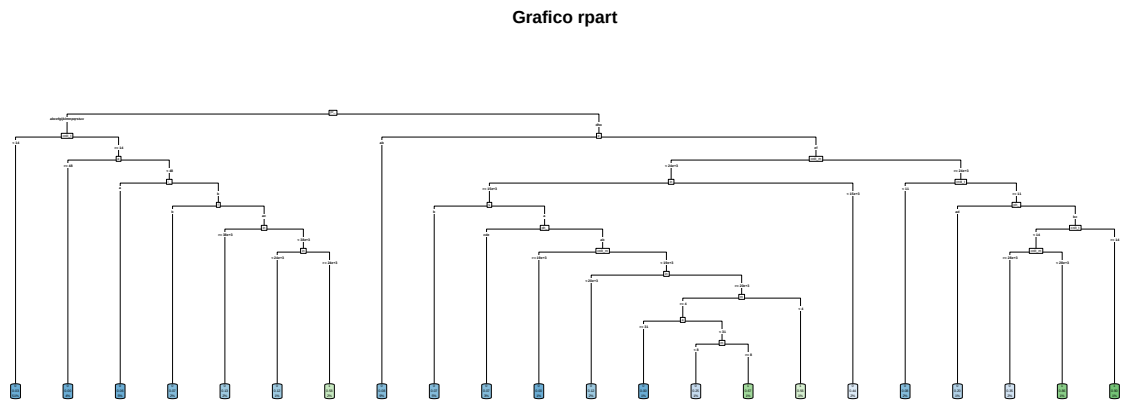
3.1.2 Grafici

Un'altra differenza tra i due pacchetti *tree* ed *rpart*, è la modalità di visualizzazione dei grafici. Nel primo caso è possibile utilizzare la funzione *plot()* assieme alla funzione *text()* per rappresentare l'albero in un modo elementare. Se invece l'albero è stato creato con la funzione *rpart*, è possibile rappresentarlo sfruttando la funzione *plot.rpart()* del pacchetto *plot.rpart*, che garantisce una maggiore comprensibilità del grafico.

```
plot(tree.model, main = "Grafico tree")
text(tree.model, pretty = 1)
```



```
rpart.plot(rpart.model, type = 5, varlen = 1, faclen = 1, main = "Grafico rpart")
```



Come è possibile constatare, una rappresentazione dell'albero non "potato" è di difficile interpretazione.

3.1.3 Test

Una volta creato l'albero è opportuno verificarne le prestazioni, è a questo punto che sarà utilizzato il subset di testing del dataset. Servendosi della funzione *predict()* presente in entrambi i pacchetti, è possibile prevedere l'output dei modelli costruiti in precedenza.

```
tree.pred <- predict(tree.model, Test.data, type = "class")
rpart.pred <- predict(rpart.model, Test.data, type = "class")
```

Per verificare il fit del modello è necessario confrontare i dati ottenuti tramite la funzione *predict()* con la colonna relativa alla variabile dipendente *bad_client_target* estratta dal subset di testing.

```
##
## tree.pred   0   1
##         0 743  94
##         1  14  11
```

```
##
## rpart.pred   0   1
##          0 711  94
##          1  46  11
```

La prestazione dei due modelli può essere valutata come percentuale di classificazioni corrette sul totale delle osservazioni.

Percentuale di classificazioni corrette del modello *tree*:

```
## [1] 0.87471
```

Percentuale di classificazioni corrette del modello *rpart*:

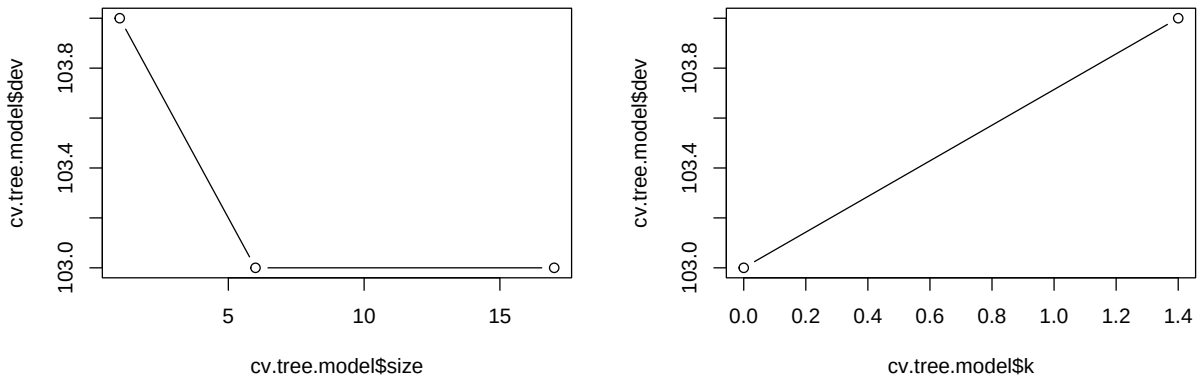
```
## [1] 0.837587
```

3.1.4 Cross-validation

Per valutare un'eventuale potatura dell'albero creato con la funzione *tree()*, è opportuno utilizzare la cross-validation con la funzione *cv.tree*.

```
set.seed(1)
cv.tree.model <- cv.tree(tree.model, FUN = prune.misclass)
```

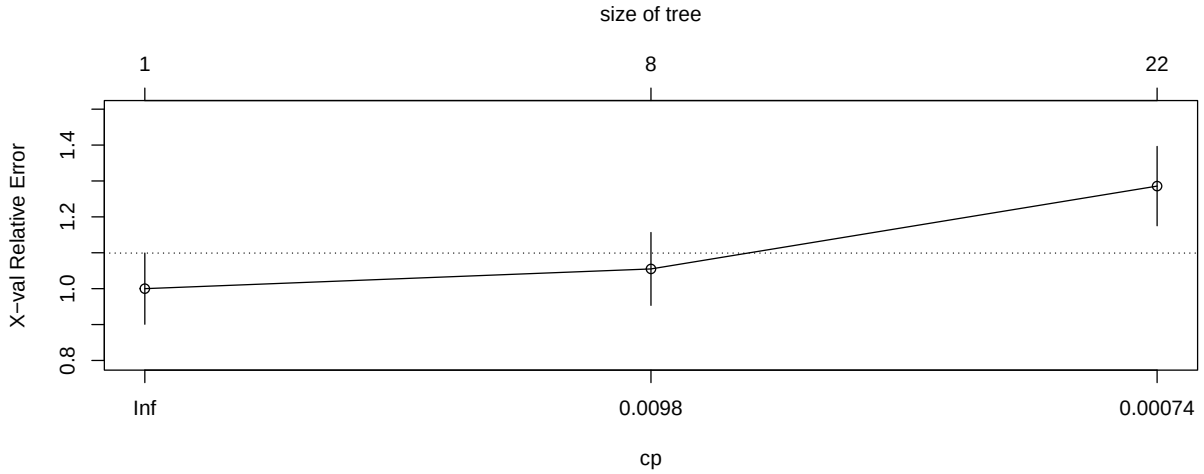
Ottenuti i risultati della cross-validation, per visualizzare meglio l'andamento delle prestazioni dell'albero in funzione della sua complessità, si può rappresentare graficamente la componente *dev* in funzione del numero di nodi dell'albero e del suo parametro di complessità *k*.



In questo caso è evidente come un numero di nodi superiore a 6 aumenta la complessità dell'albero senza però portare a nessun miglioramento nel suo potere predittivo. Questa informazione tornerà utile nella fase di potatura.

Nel caso dell'albero costruito con *rpart* non è necessario effettuare la cross-validation, in quanto già eseguita in fase di creazione dell'albero. Le valutazioni sulla dimensione ottimale possono essere fatte osservando il grafico prodotto dalla funzione *plotcp()* presente all'interno del pacchetto *rpart*, che rappresenta l'errore relativo del modello in funzione del parametro di complessità dell'albero *cp*.

```
plotcp(rpart.model)
```

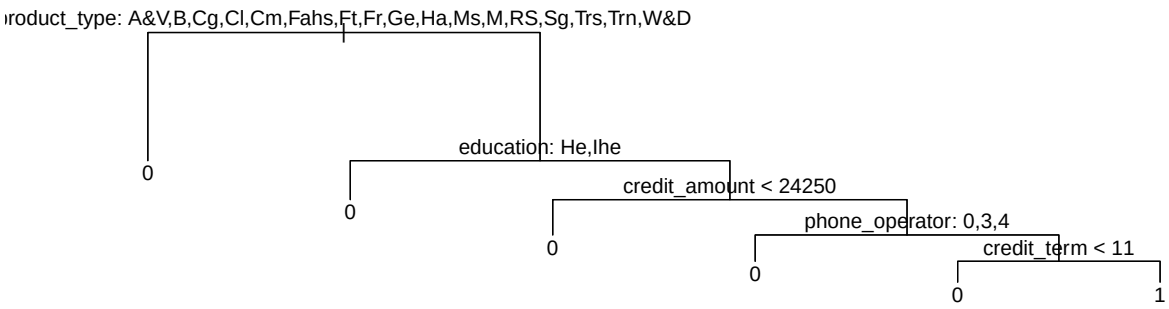


La scelta della dimensione ottimale ricade su 8 nodi, corrispondenti ad un cp di circa 0.01, per avere il giusto equilibrio tra precisione del modello e completezza.

3.1.5 Potatura

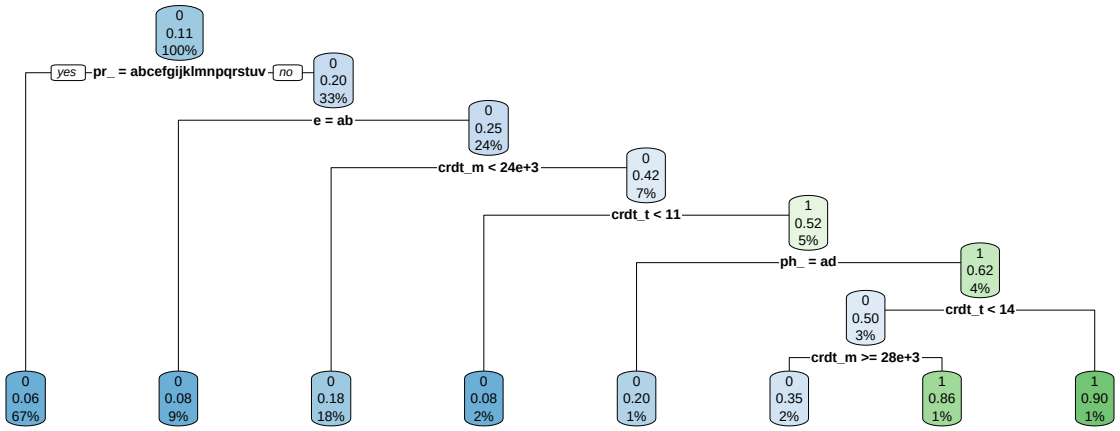
Per massimizzare le prestazioni del modello *tree* minimizzandone la complessità si procede ad una sua potatura impostando il numero di nodi a 6.

```
prune.tree.model <- prune.misclass(tree.model, best = 6)
plot(prune.tree.model)
text(prune.tree.model, pretty = 1)
```



La potatura dell'albero *rpart* invece è necessario impostare il parametro *cp* a 0.01, come emerso dal grafico *plotcp()*.

```
prune.rpart.model <- prune.rpart(rpart.model, cp = 0.01)
rpart.plot(prune.rpart.model, faclen = 1, varlen = 1)
```



Dai grafici è evidente come, effettuando il pruning sui due alberi,la chiarezza e l'interpretabilità dei due modelli migliorino.

3.1.6 Test

Una volta potati gli alberi, è opportuno verificarne le prestazioni come fatto in precedenza.

```
##
## tree.pred1   0   1
##          0 743  94
##          1  14  11
```

```
##
## rpart.pred1   0   1
##           0 750 102
##           1   7   3
```

Percentuale di classificazioni corrette del modello "tree" post pruning:

```
## [1] 0.87471
```

Percentuale di classificazioni corrette del modello "rpart" post pruning:

```
## [1] 0.8735499
```

Il risultato può sembrare peggiore nel secondo caso ma, analizzando meglio le tabelle delle classificazioni, vediamo come calano i falsi negativi *[0,1]*, ovvero quei clienti che sono considerati come buoni pagatori ma che in realtà non lo sono. Questo rende il modello meno corretto ma più cautelativo, il che torna a beneficio dell'utilizzatore vista l'applicazione.

L'adattamento del modello alle necessità è stato ottenuto grazie alla maggiore elasticità garantita dai parametri della funzione *rpart*.

3.2 Regressione

Il dataset utilizzato per l'albero di regressione è *concrete*. Quest'ultimo raccoglie i dati di 1030 campioni di cemento di cui riporta le seguenti informazioni suddivise in 9 colonne:

- *cement*: *kg* di cemento per m^3 di miscela (variabile quantitativa);
- *slag*: *kg* di scorie di altoforno per m^3 di miscela (variabile quantitativa);
- *flyash*: *kg* di ceneri volanti per m^3 di miscela (variabile quantitativa);
- *water*: *kg* di acqua per m^3 di miscela (variabile quantitativa);
- *superplasticizer*: *kg* di fluidificante per m^3 di miscela (variabile quantitativa);
- *coarseaggregate*: *kg* di aggregato grosso per m^3 di miscela (variabile quantitativa);
- *fineaggregate*: *kg* di aggregato fine per m^3 di miscela (variabile quantitativa);
- *age*: età del campione di calcestruzzo (variabile quantitativa);
- *csMPa*: resistenza alla compressione del calcestruzzo in *MPα* (variabile dipendente quantitativa).

L'obiettivo è sviluppare un modello in grado di prevedere la resistenza alla compressione del campione di calcestruzzo, *csMPa*, in funzione delle restanti variabili.

Anche in questo caso è opportuno suddividere il dataset in due subset, training e testing.

3.2.1 Training

Il procedimento di training dei modelli è lo stesso già utilizzato nel capitolo relativo agli alberi di classificazione.

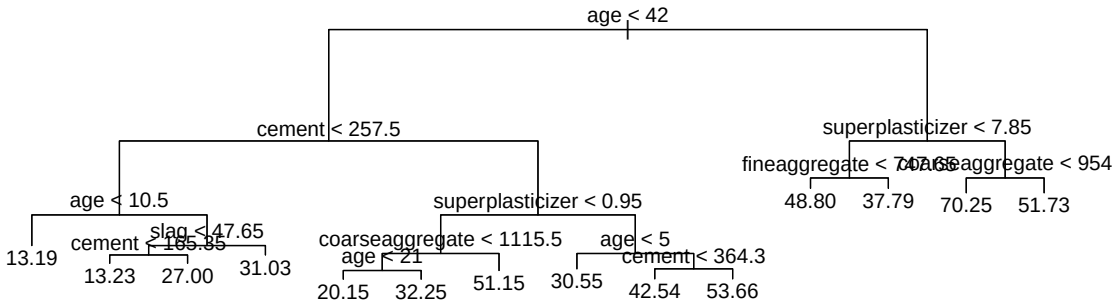
```
tree.concrete <- tree(csMPa ~ ., Train.Data,
                      control = tree.control(nrow(Train.data),
                                             mincut = 5,
                                             minsize = 10,
                                             mindev = 0.01))
```

```
rpart.concrete <- rpart(csMPa ~ ., Train.Data,
                        control = rpart.control(minbucket = round(20/3),
                                                cp = 0.0001,
                                                maxcompete = 6,
                                                maxsurrogate = 6,
                                                usesurrogate = 1,
                                                xval = 100,
                                                surrogatestyle = 1,
                                                maxdepth = 20))
```

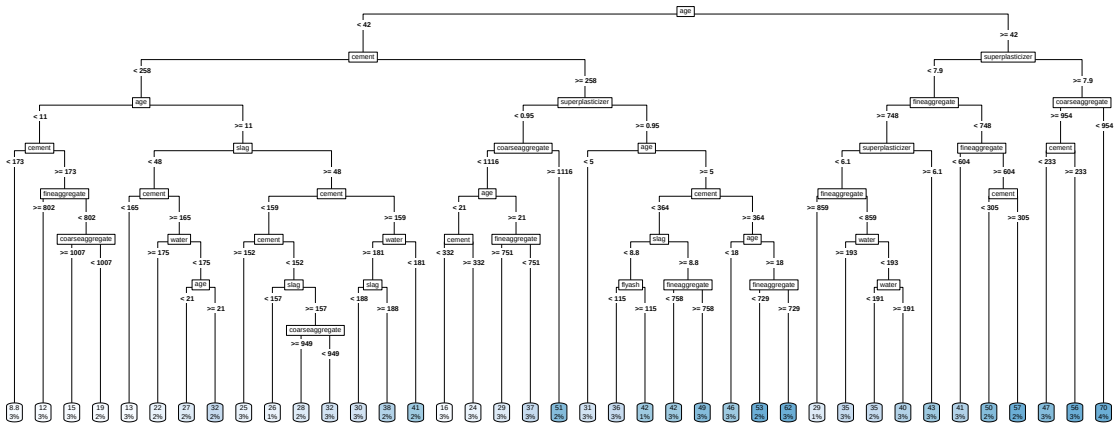
3.2.2 Grafici

Anche in questo caso è possibile vedere la differenza nella rappresentazione grafica dei due alberi *tree* e *rpart*.

```
plot(tree.concrete)
text(tree.concrete, pretty = 1)
```




```
rpart.plot(rpart.concrete, type = 5)
```



3.2.3 Cross-validation

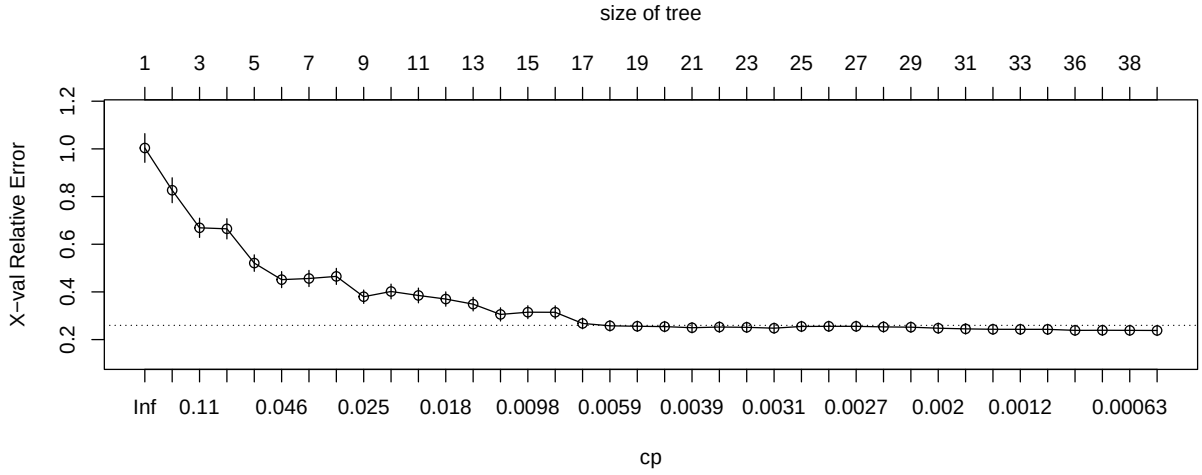
Come nel caso precedente, per l'albero costruito con la funzione *tree* va effettuata la cross-validation, mentre per l'albero *rpart* ciò non è necessario per poter fare le dovute valutazioni in merito alla fase di pruning.

```r
set.seed(1)
cv.tree.concrete <- cv.tree(tree.concrete)
```

```r
plot(cv.tree.concrete$size, cv.tree.concrete$dev, type = "b")
```

L'albero *tree* può dare buoni risultati già alla dimensione di 9 nodi. Oltre si otterrebbe un miglioramento nella capacità predittiva relativamente basso rispetto all'incremento del numero di nodi.

```r
plotcp(rpart.concrete)
```

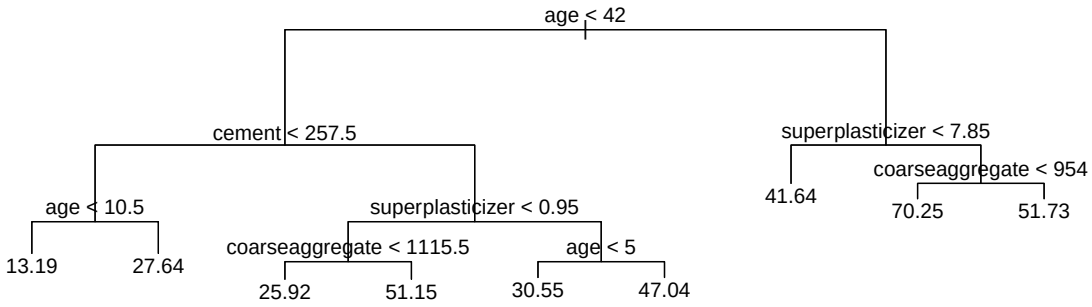


La funzione *plotcp()* permette di avere un orizzonte di valutazione superiore sull'andamento della capacità predittiva dell'albero *rpart* in funzione del suo parametro di complessità *cp*, infatti è possibile vedere che a 17 nodi l'accuratezza del modello si stabilizza.

3.2.4 Potatura

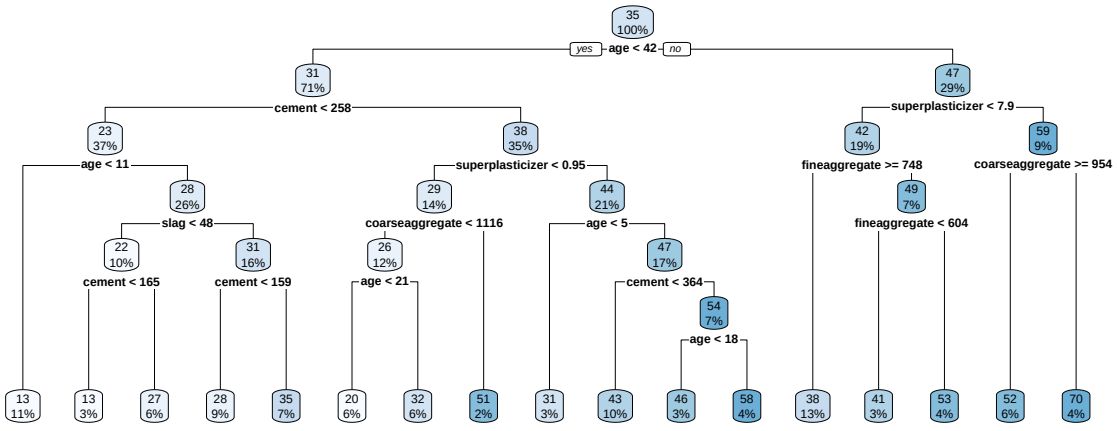
Il pruning dell'albero *tree* è stata effettuata a 9 nodi.

```
prune.tree.concrete <- prune.tree(tree.concrete, best = 9)
plot(prune.tree.concrete)
text(prune.tree.concrete, pretty = 1)
```



Il pruning dell'albero *rpart* è stato effettuato impostando $cp = 0.007$.

```
prune.rpart.concrete <- prune.rpart(rpart.concrete, cp = 0.007)
rpart.plot(prune.rpart.concrete)
```



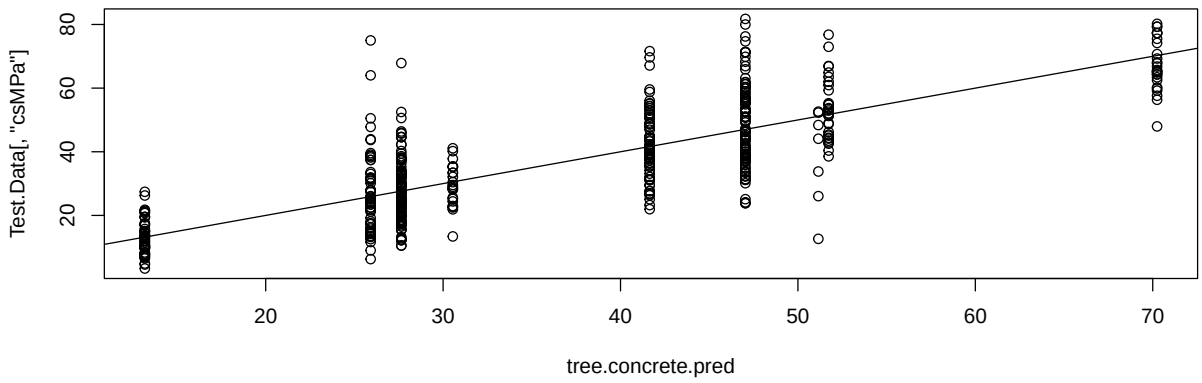
3.2.5 Test

Come nel caso della classificazione, anche per gli alberi di regressione è possibile utilizzare la funzione *predict()* per testare i modelli.

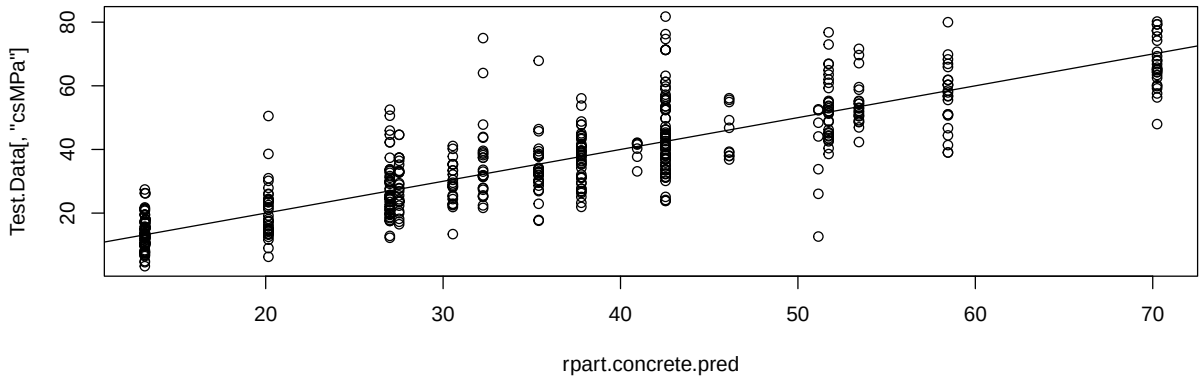
```
tree.concrete.pred <- predict(prune.tree.concrete, Test.Data)
rpart.concrete.pred <- predict(prune.rpart.concrete, Test.Data)
```

Per valutare il fitting dei due modelli in questo caso si procede valutando lo scostamento dei valori previsti rispetto ai valori del subset di test.

```
plot(tree.concrete.pred, Test.Data[, "csMPa"])
abline(0,1)
```




```
plot(rpart.concrete.pred, Test.Data[, "csMPa"])
abline(0,1)
```



Il modello *rpart()* ha un fit migliore rispetto a quello costruito con la funzione *tree()*.

```r
sqrt(mean((tree.concrete.pred - Test.Data[, "csMPa"])^2))
```

```
## [1] 10.59384
```

```r
sqrt(mean((rpart.concrete.pred - Test.Data[, "csMPa"])^2))
```

```
## [1] 9.396566
```

3.3 Confronto

La comparazione tra le funzionalità comuni ai due pacchetti, *tree* e *rpart*, può essere fatta su tre livelli:

- *Creazione del modello*: *rpart* permette di personalizzare il processo di creazione dell'albero in maniera più puntuale rispetto a quello che è possibile fare con il pacchetto *tree*;
- *Fitting del modello*: *rpart* fornisce soluzioni più pratiche e informazioni più complete a sostegno della fase di potatura dell'albero;
- *Rappresentazione grafica del modello*: *rpart* permette di rappresentare i grafici degli alberi creati in maniera più chiara e di semplice applicazione rispetto al pacchetto *tree* il quale invece offre soluzioni essenziali e meno user friendly.

4 Conclusioni

Il presente studio si è focalizzato sull'utilizzo di due importanti pacchetti disponibili in R, il pacchetto *tree* ed il pacchetto *rpart*.

Entrambi i pacchetti sono stati sviluppati sulla base del libro e degli algoritmi *CART (Classification And Regression Tree)* del 1984 di Breiman, Friedman, Olshen e Stone.

Ne emerge che sono pacchetti ottimi per lo sviluppo di alberi decisionali, i quali sono utili in diversi ambiti, come il data mining, il machine learning oppure il knowledge discovery. Tutto ciò è possibile grazie alla chiara modellazione ed alla semplicità di interpretazione nella fase di rappresentazione visiva.

Secondo il Prof. Brian Ripley, autore del pacchetto *tree*, *rpart* lavora maggiormente a livello di linguaggio C, incluso il pruning e la cross-validation, rendendolo molto più veloce. Invece, *tree* è molto più vicino alle funzionalità del linguaggio di programmazione S ed è disponibile in R per supportare il libro MASS (Modern Applied Statistics with S) del Prof. Ripley.

In conclusione, il pacchetto *tree* pone le basi per la gestione di alberi decisionali per la classificazione e la regressione. Tuttavia, il pacchetto *rpart* permette di svolgere le stesse attività che è possibile svolgere con il pacchetto *tree* in maniera ottimizzata e più accurata, oltre a portare con sé funzionalità aggiuntive.

5 Sitografia

Di seguito viene riportata la sitografia:

Alberi di Classificazione (Classification Trees: CTREE)
Decision Tree: cos'è e come funziona
Gli Alberi decisionali (Machine Learning)
R Decision Tree tutoirial
Regression Tree - IBM Documentation
Classification Tree - IBM Documentation
RDocumentation - tree
RDocumentation - rpart
Chapter 26 Trees | R for Statistical Learning
A comparison on using R.tree vs R.rpart
R Difference between "tree" and "rpart"