Data Mining e Bioimmagini

Classificazione in R, Algoritmi di classificazione, alberi decisionali, ConfusionMatrix, Makefile

Prof. Pierangelo Veltri – 16/10/2023- Autori: Maturo, Accetturo - Revisionatori: Accetturo

Classificazione in R

La classificazione in R è una tecnica di apprendimento automatico che consiste nell'assegnare automaticamente un'etichetta o una classe a un insieme di dati in base a modelli e algoritmi costruiti da dati di addestramento.
R offre numerose librerie e pacchetti per l'implementazione di algoritmi di classificazione e utilizzeremo il pacchetto "Caret", mentre per gli alberi decisionali utilizzeremo "rpart".

#Come si installa Caret:
Scrivere in R la seguente riga di codice per procedere all'installazione
" *install.packages("caret")* "

Quando si dovrà utilizzare la libreria basterà richiamarla in R scrivendo
" library(caret) "

I passaggi da eseguire per la classificazione in R sono:

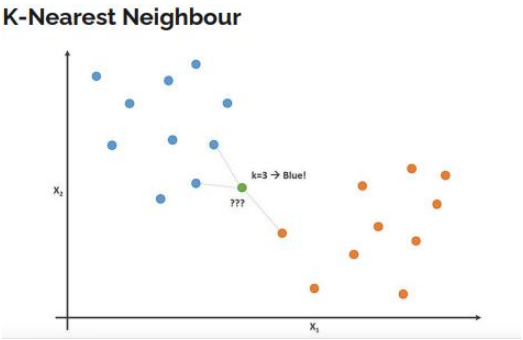
1. **Caricamento del Dataset**
   Come esempio utilizzeremo il dataset "Iris" e la riga di codice che andremo a scrivere sarà:
   " *data(iris*) "
   Con la funzione " *head(iris*) " il dataset ci verrà riportato in righe e colonne per avere un idea più chiara del suo contenuto.
   Iris è un dataset che descrive una collezione di 150 piante tramite varie misure (features) e ogni pianta è un esempio, descritto dalle seguenti misure: lunghezza del sepalo, larghezza del sepalo, lunghezza del petalo, larghezza del petalo.

2. **Divisione del dataset in Trainingset e Testset**
   La suddivisione del dataset in un training set e un test set è una parte cruciale del processo di classificazione in modo che sia possibile valutare le prestazioni del modello.
   Prima di suddividere in modo casuale il dataset appena caricato in training set e testset, fissiamo il seed con **set.seed()** per garantire la riproducibilità dei risultati (questo passaggio è opzionale ma utile quando si desidera ottenere gli stessi risultati in esecuzioni successive).
   A seguire si procede con la suddivisione dove per esempio si prenderà il 70% per il trainingset e il 30% per il testset

```
trainIndex <- createDataPartition(iris$Species, p = 0.7, list = FALSE)
trainData <- iris[trainIndex, ]
testData <- iris[-trainIndex, ]
```

   Il training set viene creato selezionando le osservazioni con gli indici generati, mentre il test set viene creato selezionando le osservazioni complementari (ossia quelle non presenti nel training set).
   Puoi passare il training set come argomento alla funzione **train()** e utilizzare il test set per valutare il modello.

3. **Selezione dell'algoritmo di classificazione**
   Si utilizza la funzione "train" di caret per selezionare l'algoritmo di classificazione desiderato.
   Ad esempio, puoi utilizzare il support vector machine (SVM) o il k-Nearest Neighbors (k-NN).
   La sintassi di base è la seguente:
   " *model <- train(Class ~ ., data = training_data, method = "method_name")* "
   Assicurati di sostituire "Class" con il nome della variabile di classe nel tuo dataset e "method_name" con il nome dell'algoritmo di classificazione scelto.



4. **Addestramento del modello (training set)**
   Come modello per l'addestramento si utilizzerà l'algoritmo k-nearest neighbors che considera le feature dei punti per cercare i k punti vicini che più assomigliano al corrente. Un "punto" nel dataset iris è uno dei fiori.
   Come k solitamente si sceglie un numero intero positivo non molto grande, la sua scelta dipende fortemente dal problema.
   In R scriveremo:
   " *model <- train(Species ~ ., data = trainData, method = "knn")* "
   dove "Species" è la feature di classe, i dati di addestramento sono nella variabile trainData e il metodo + "knn

   La scelta di k?
   L'addestramento con la funzione train( ) della libreria caret tratta il parametro K del modello come un iperparametro da esplorare.
   Infatti, se facciamo stampare la variabile model otterremo i dettagli (immagine sulla destra).
   Si vede come varia l'accuratezza del modello al variare del parametro K:
   R sceglierà la migliore

```
model

k  Accuracy   ...
5  0.9597886  ...
7  0.9615954  ...
9  0.9658448  ...
```

5. **Valutazione del modello**
   Si verifica il comportamento del modello sul testset.
   Si ricorda che questa è la reale misura dell'accuratezza, in quanto i dati contenuti nel testset non sono mai stati visti dal modello durante l'addestramento.
   La sintassi è la seguente:
   " *predictions <- predict(model, newdata = testData)* "
   A seguire si misurano le performance di accuratezza usando la matrice di confusione la quale sintassi è:
   " *accuracy <- confusionMatrix(predictions, testData$Species)$overall["Accuracy*"] "

   Nel nostro caso l'accuratezza raggiunta dal modello sarà pari a 0.9555556

6. **Misurazione delle performance (Matrice di confusione)**
   La matrice di confusione è uno strumento utile per valutare le prestazioni di un modello di classificazione.
   Essa mostra quanti campioni sono stati classificati correttamente e quante previsioni errate sono state fatte ed è particolarmente utile in problemi di classificazione binaria, ma può essere adattata per problemi di classificazione multiclasse.

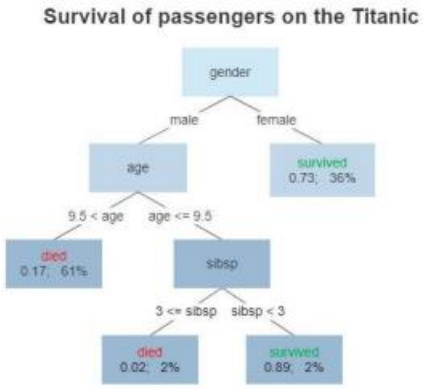
   Il comando per accedervi è:
   " *confusionMatrix(predictions, testData$Species)* "
   che, fra le altre, ci restituirà Confusion Matrix and Statistics
   Sulla colonna sinistra troviamo le predizione fatte dal modello che incrociati con la riga che riporta la tipologia del dato reale si verifica se il modello ha azzeccato il dato o ha errato riportando anche la quantità di volte.
   Il NIR è il No Association Rate e indica l'accuratezza che si può ottenere predicendo sempre la classe più numerosa

	Reference		
Prediction	setosa	versicolor	virginica
setosa	15	0	0
versicolor	0	14	1
virginica	0	1	14

Numero di istanze predette nella classe corretta o nella classe errata

Alberi di Decisione

Descrizione del modello
Gli alberi di decisione sono modelli usati sia in statistica ma più spesso nel data mining.
Per ogni variabile si cerca di capire se sia rilevante per dividere il dataset nelle classi note ○ quanto la variabile "spieghi" le classi
Se è sufficientemente rilevante, viene stabilita una soglia di partizionamento, e aggiunti due nuovi nodi all'albero ○ se il valore è minore della soglia, si percorrerà il sotto-albero sinistro ○ se maggiore della soglia, il sottoalbero destro.



Proseguiremo con un esempio dove utilizzeremo di nuovo Iris e l'unica differenza con il precedente sarà la libreria, dove in questo caso utilizzeremo "rpart"

Processo:

1. La scelta del dataset e del modello da utilizzare sarà uguale come descritto antecedentemente
   La sintassi per l'addestramento sarà
   dt <- rpart(Species~., data = trainData, method = "class")

   Se a fine addestramento si vuole avere una versione testuale dell'albero bisognerà digitare

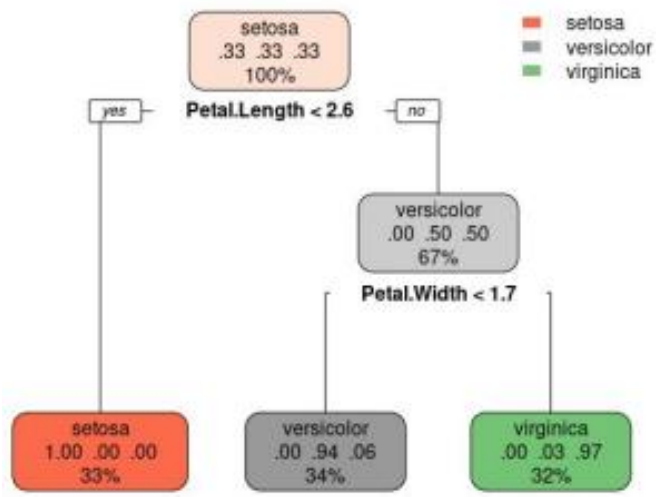
"dt" e la risposta sarà la seguente:

```
n= 105                                          addestramento effettuato su 105 esempi
node), split, n, loss, yval, (yprob)
      * denotes terminal node                   nodi foglia indicati con un asterisco
1) root 105 70 setosa (0.33333333 0.33333333 0.33333333)
  2) Petal.Length< 2.6 35  0 setosa (1.00000000 0.00000000 0.00000000) *
  3) Petal.Length>=2.6 70 35 versicolor (0.00000000 0.50000000 0.50000000)
    6) Petal.Width< 1.65 36  2 versicolor (0.00000000 0.94444444 0.05555556) *
    7) Petal.Width>=1.65 34  1 virginica (0.00000000 0.02941176 0.97058824) *
```

Si può avere anche una versione grafica del risultato ma prima bisognerà installare una libreria
" *install.packages("rpart.plot*") "

e quindi possiamo usarla con:
" *library(rpart.plot)* "

Finalmente possiamo procedere alla generazione del grafico dell'albero di decisione:
" *rpart.plot(dt)* "



2. La valutazione del modello sarà uguale alla antedecente con la confusionMatrix e la sintassi sarà:
   predictions <- predict(dtm, testData, type="class")

Makefile

Il Makefile è un file di tipo script. Il comando "make" legge un file contenente le istruzioni da eseguire e le esegue in base a quanto richiesto.

Ad esempio, immaginiamo di voler costruire un sistema decisionale. Un programmatore si occupa della creazione degli alberi e scrive una serie di comandi, un altro contribuisce con un'altra parte, e così via. Per ottenere un sistema completo, è necessario compilare e collegare tra loro queste diverse parti. La compilazione implica la creazione di un eseguibile specifico per la macchina in uso, ad esempio Unix o Windows.

Sistemi operativi

Se devo installare un'applicazione su iOS, non posso farlo attraverso Google Play poiché iOS e Android sono due sistemi operativi diversi con comandi incompatibili. I sistemi operativi forniscono l'ambiente in cui i programmi possono essere eseguiti, ma su iOS e Android, i comandi sono differenti. Nonostante ciò, l'obiettivo finale di entrambi è lo stesso: ad esempio, entrambi eseguono il comando "print".

Pertanto, le applicazioni per questi sistemi operativi possono avere funzionalità simili ma sono sviluppate in ambienti diversi. Ecco perché il Makefile è un comando che invoca i compilatori specifici per il sistema operativo in uso. Se il Makefile manca, come nel tuo caso, l'applicazione non può essere sviluppata poiché non trova il comando necessario.

Per esempio, quando si installa Instagram, indipendentemente dal dispositivo (Samsung o iPhone), l'applicazione richiede l'autorizzazione per accedere alla fotocamera. Ciò implica che lo sviluppatore ha creato un'applicazione compatibile con entrambi i sistemi operativi, ma entrambi i sistemi devono offrire un pulsante per richiedere l'autorizzazione alla fotocamera.

Quando si installa una nuova libreria, possono verificarsi due scenari: funziona correttamente o si verificano errori. Nel secondo caso, è necessario individuare il problema all'interno della sequenza di comandi.

Per esempio, se stai installando un'applicazione su un dispositivo non aggiornato, potrebbe essere necessario aggiornarlo affinché l'app funzioni. L'applicazione si aspetta che sia disponibile un comando specifico che richiede due parametri per funzionare. Questo rappresenta l'importanza di comprendere come utilizzare i comandi all'interno di una sequenza, anche se non si comprende appieno il loro funzionamento.

In sintesi, i sistemi operativi sono sviluppati da meccanismi diversi, ma le interfacce devono rimanere consistenti. Java è stato noto per la sua indipendenza dalla piattaforma, traducendo i comandi standard in strutture compatibili con vari sistemi operativi. Ad esempio, Chrome, una volta installato, è in grado di tradurre le richieste di accesso alla RAM in modo compatibile con il sistema operativo.

Di conseguenza, le API (Application Programming Interface), più ampie e generiche sono, meglio possono funzionare per garantire la compatibilità tra diversi sistemi.