ALGORITHMS USED IN R

Content

1. LINEAR REGRESSION:

Regression analysis is a very widely used statistical tool to establish a relationship model between two variables. One of these variables is called **predictor** variable whose value is gathered through experiments. The other variable is called the **response** variable whose value is derived from the predictor variable.

Mathematically a linear relationship represents a straight line when plotted as a graph. The general mathematical equation for a linear regression is −

$$y = ax + b$$

where;

y is the response variable.

x is the predictor variable.

a and b are constants which are called the coefficients.

Steps to Establish a Regression in R

A simple example of regression is predicting weight of a person when his height is known. To do this we need to have the relationship between height and weight of a person.

The steps to create the relationship is −

- Experiment with gathering a sample of observed values of height and corresponding weight.
- Create a relationship model using the lm() functions in R.
- Find the coefficients from the model created and create the mathematical equation using these
- Get a summary of the relationship model to know the average error in prediction. Also called residuals.
- To predict the weight of new persons, use the predict() function in R.

Input Data:

Below is an example of a dataset consisting of different values of height and weight

# Values of height:

151, 174, 138, 186, 128, 136, 179, 163, 152, 131

# Values of weight:

63, 81, 56, 91, 47, 57, 76, 72, 62, 48

Lm() Function:

This function creates the relationship model between the predictor and the response variable.

Syntax: The basic syntax for the lm() function in linear regression is −

lm(formula,data)

where,

The formula is a symbol presenting the relation between x and y, and,

Data is a vector quantity to which the formula will be applied.

Creating Relationship Models and Getting Coefficients:

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

# Apply the lm() function:

relation <- lm(y~x)

print(relation)

When we execute and call the above code, we get,

Call:

lm(formula = y ~ x)

```
Coefficients:
(Intercept)          x
  -38.4551         0.6746
```

In the above data , we got the intercept of y and coefficient of x. Now to get the data summary, we will use print(summary(relation)). The following code is shown below:

print(summary(relation))

When we execute the above code, we get the following results:

```
Residuals:
   Min      1Q     Median      3Q     Max
-6.3002    -1.6629  0.0412    1.8944  3.9775

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -38.45509    8.04901  -4.778  0.00139 **
x            0.67461    0.05191  12.997 1.16e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 3.253 on 8 degrees of freedom
Multiple R-squared:  0.9548,    Adjusted R-squared:  0.9491
F-statistic: 168.9 on 1 and 8 DF,  p-value: 1.164e-06
```

Predict() Function:

The basic syntax for predict() in linear regression is :

predict(object, newdata)

Where;

The object is the formula that is already created using the lm() function.

Newdata is the vector containing the new value for the predictor variable.

Predicting the weight of new persons:

```
# The predictor vector.
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
# The response vector.
y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
# Apply the lm() function.
relation <- lm(y~x)

# Find weight of a person with height 170.
a <- data.frame(x = 170)
result <-  predict(relation,a)
print(result)
```

When executed, it gives the following results:

```
       1
76.22869
```

If we visualise this scenario graphically then we get;

```
# Give the chart file a name.
png(file = "linearregression.png")

# Plot the chart.
plot(y,x,col = "blue",main = "Height & Weight Regression",
  abline(lm(x~y)),cex = 1.3,pch = 16,xlab = "Weight in Kg",ylab = "Height in cm")

# Save the file.
dev.off()
```

When the above code is executed , the graph obtained is shown below:

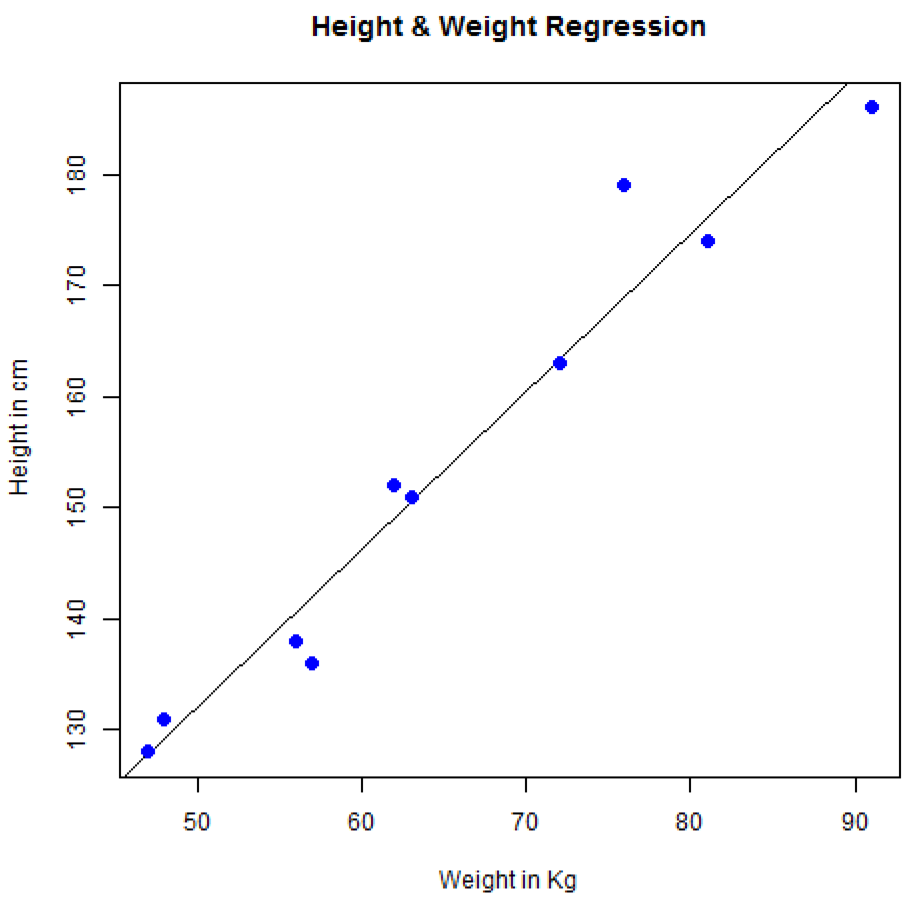


Fig-1 Graph showing the relationship bw height and weight using LR

2. LOGISTIC REGRESSION

The Logistic Regression is a regression model in which the response variable (dependent variable) has categorical values such as True/False or 0/1. It actually measures the probability of a binary response as the value of response variable based on the mathematical equation relating it with the predictor variables.

The general mathematical equation for logistic regression is −

y = 1/(1+e^-(a+b1x1+b2x2+b3x3+...))

Where;

y is the response variable.

x is the predictor variable.

a and b are the coefficients which are numeric constants.

The function used to create the regression model is the **glm()** function.

Syntax: The basic syntax for glm() function in logistic regression is :

glm(formula,data,family)

Where,

Formula is the symbol presenting the relationship between the variables.

Data is the data set giving the values of these variables.

Family is R object to specify the details of the model. It's value is binomial for logistic regression.

We will understand this concept with the help of an example:

The dataset that we will choose is a built-in dataset called "mtcars" which describes different models of a car with their various engine specifications. In "mtcars" data set, the transmission mode (automatic or manual) is described by the column am which is a binary value (0 or 1). We can create a logistic regression model between the columns "am" and 3 other columns - hp, wt and cyl.

Establishing the code:

```
# Select some columns form mtcars.
input <- mtcars[,c("am","cyl","hp","wt")]
```

print(head(input))

When we execute the above code, it produces the following result −

```
                  am   cyl  hp    wt
Mazda RX4          1   6    110   2.620
Mazda RX4 Wag      1   6    110   2.875
Datsun 710         1   4     93   2.320
Hornet 4 Drive     0   6    110   3.215
Hornet Sportabout  0   8    175   3.440
Valiant            0   6    105   3.460
```

Creating the regression model:

We use the glm() function to create the regression model and get its summary for analysis.

am.data = glm(formula = am ~ cyl + hp + wt, data = input, family = binomial)

print(summary(am.data))

When we execute the above code, it produces the following result −

```
Call:
glm(formula = am ~ cyl + hp + wt, family = binomial, data = input)

Deviance Residuals:
    Min        1Q      Median       3Q       Max
-2.17272    -0.14907  -0.01464    0.14116   1.27641

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 19.70288    8.11637   2.428   0.0152 *
cyl          0.48760    1.07162   0.455   0.6491
hp           0.03259    0.01886   1.728   0.0840 .
wt          -9.14947    4.15332  -2.203   0.0276 *
```

```
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 43.2297  on 31  degrees of freedom
Residual deviance:  9.8415  on 28  degrees of freedom
AIC: 17.841

Number of Fisher Scoring iterations: 8
```

In the summary as the p-value in the last column is more than 0.05 for the variables "cyl" and "hp", we consider them to be insignificant in contributing to the value of the variable "am". Only weight (wt) impacts the "am" value in this regression model.

3. DECISION TREE

Decision tree is a graph to represent choices and their results in form of a tree. The nodes in the graph represent an event or choice and the edges of the graph represent the decision rules or conditions. It is mostly used in Machine Learning and Data Mining applications using R.
Generally, a model is created with observed data also called training data. Then a set of validation data is used to verify and improve the model. R has packages which are used to create and visualize decision trees. For new set of predictor variable, we use this model to decide the data.
The R package **"party"** is used to create decision trees.

Getting Started with Decision Trees in R

Installing R Packages:

We use the below command in the R console to install the package. We also have to install the dependent packages if any.

install.packages("party")

The package "party" has the function ***ctree()*** used to create and analyze decision trees.
Syntax: The basic syntax for creating a decision tree in R is −

ctree(formula, data)

Where;
A formula is a formula describing the predictor and response variables.
Data is the name of the data set used.

Input Data:
We will use the R in-built data set named readingSkills to create a decision tree. It describes the score of someone's readingSkills if we know the variables "age","shoesize","score" and whether the person is a native speaker or not. The same dataset was used for Randomforest too.
The sample data is as follows:
# Load the party package. It will automatically load other
# dependent packages.

library(party)

# Print some records from data set readingSkills.

print(head(readingSkills))

When we execute the above code, we get the following results;

```
  nativeSpeaker   age   shoeSize      score
 1           yes     5   24.83189   32.29385
```

```
2          yes    6   25.95238   36.63105
3           no   11   30.42170   49.60593
4          yes    7   28.66450   40.28456
5          yes   11   31.88207   55.46085
6          yes   10   30.07843   52.83124
Loading required package: methods
Loading required package: grid
```

We will use the **ctree()** function to create the decision tree and see its graph.

```
# Create the input data frame.
input.dat <- readingSkills[c(1:105),]

# Give the chart file a name.
png(file = "decision_tree.png")

# Create the tree.
output.tree <- ctree(
nativeSpeaker ~ age + shoeSize + score,
data = input.dat)

# Plot the tree.
plot(output.tree)

# Save the file.
dev.off()
```

When we execute the above code, it produces the following result −

```
null device
          1
Loading required package: methods
Loading required package: grid
Loading required package: mvtnorm
Loading required package: modeltools
Loading required package: stats4
Loading required package: strucchange
Loading required package: zoo

Attaching package: ‘zoo’
```

The following objects are masked from ‘package:base’:

as.Date, as.Date.numeric

Loading required package: sandwich

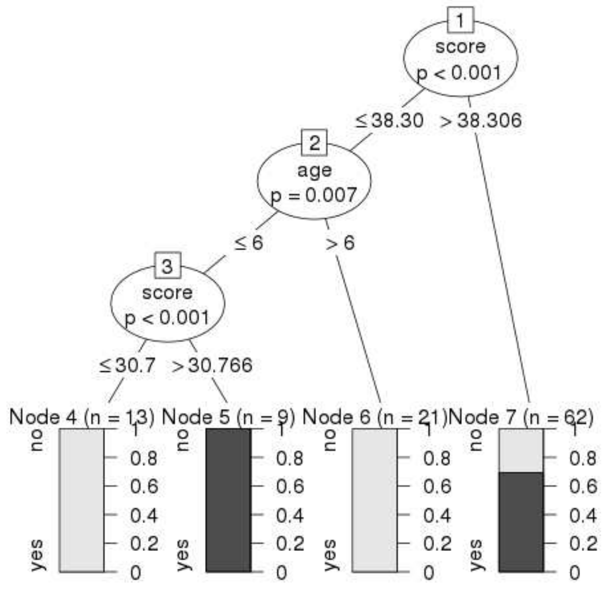


Fig-2 Decision Tree graph

From the decision tree shown above we can conclude that anyone whose readingSkills score is less than 38.3 and age is more than 6 is not a native Speaker.

4. RANDOM FOREST

In the random forest approach, a large number of decision trees are created. Every observation is fed into every decision tree. The most common outcome for each observation is used as the final output. A new observation is fed into all the trees and taking a majority vote for each classification model.
An error estimate is made for the cases which were not used while building the tree.

That is called an **OOB (Out-of-bag) error estimate** which is mentioned as a percentage.
The R package **"randomForest"** is used to create random forests.

Getting Started with Random Forest in R
Install R package
Use the below command in R console to install the package. We also have to install the dependent packages if any.

install.packages("randomForest)

The package "randomForest" has the function **randomForest()** which is used to create and analyze random forests.
Syntax: The basic syntax for creating a random forest in R is −

randomForest(formula, data)

Where;
A formula is a formula describing the predictor and response variables.
Data is the name of the data set used.

Input Data:
The dataset used will be a built-in dataset in R named readingSkills to create a decision tree. It describes the score of someone's readingSkills if we know the variables "age","shoe size","score" and whether the person is a native speaker.

Here is the sample data.
# Load the party package. It will automatically load other
# required packages.

library(party)

# Print some records from data set readingSkills.

print(head(readingSkills))

When the above code is followed , we get:

```
  nativeSpeaker  age   shoeSize      score
1           yes     5  24.83189   32.29385
2           yes     6  25.95238   36.63105
3            no    11  30.42170   49.60593
4           yes     7  28.66450   40.28456
5           yes    11  31.88207   55.46085
6           yes    10  30.07843   52.83124
Loading required package: methods
Loading required package: grid
```

We will use the **randomForest() function** to create the decision tree and see it's graph.

```
# Create the forest.
    output.forest <- randomForest(nativeSpeaker ~ age + shoeSize + score,
                                  data = readingSkills)

# View the forest results.
print(output.forest)

# Importance of each predictor.
print(importance(fit,type = 2))
```

When the above code is executed , we get the following results:

```
Call:
 randomForest(formula = nativeSpeaker ~ age + shoeSize + score,
        data = readingSkills)
              Type of random forest: classification
                    Number of trees: 500
No. of variables tried at each split: 1

        OOB estimate of  error rate: 1%
Confusion matrix:
    no yes class.error
no  99   1        0.01
yes  1  99        0.01
         MeanDecreaseGini
age              13.95406
shoeSize         18.91006
score            56.73051
```

From the random forest shown above we can conclude that the shoesize and score are the important factors deciding if someone is a native speaker or not. Also the model has only **1% error** which means **we can predict with 99% accuracy.**

5. NAIVE BAYES CLASSIFIER

The Naive Bayes algorithm is a classification algorithm based on Bayes' theorem. The algorithm assumes that the features are independent of each other, which is why it is called "naive." It calculates the probability of a sample belonging to a particular class based on the probabilities of its features.

In Bayesian classification, the main interest is to find the posterior probabilities i.e. the probability of a label given some observed features, **P(𝐿L | features).** With the help of Bayes theorem, we can express this in quantitative form as follows −

P(L|features)= P(L)P(features|L)/ P(features)

Where;
P(L|features) is the posterior probability of class.
P(L) is the prior probability of class.
P(features|L) is the likelihood which is the probability of predictor given class.
P(features) is the prior probability of predictor.

In the Naive Bayes algorithm, we use Bayes' theorem to calculate the probability of a sample belonging to a particular class. We calculate the probability of each feature of the sample given the class and multiply them to get the likelihood of the sample belonging to the class. We then multiply the likelihood with the prior probability of the class to get the posterior probability of the sample belonging to the class. We repeat this process for each class and choose the class with the highest probability as the class of the sample.

Building a Naive Bayes Classifier in R:
Now, let us build a Naive Bayes classifier in R using the following steps:

Step 1 - Import necessary Libraries
In this step, we will import the three essential R packages - **mlbench, caret, and e1071** for building a Naive Bayes classifier in R. If these packages are not already installed, we can easily do so with the following commands:

```
#installing required packages
install.packages('mlbench')
install.packages("caret", dependencies = c("Depends", "Suggests"))
install.packages('e1071')
```

Once the packages are successfully installed, we can load them in our R environment as shown below:

```
# importing the required libraries
```

library(mlbench)
library(caret)
library(e1071)

The mlbench package provides us with various datasets that can be used to evaluate the performance of machine learning algorithms. On the other hand, we can use the caret package for building and evaluating predictive models. Additionally, the e1071 package includes functions for training a Naive Bayes classifier.

Step 2 - Read and Explore the Dataset
We will use the built-in "Sonar" dataset from the "mlbench" package in R to build a Naive Bayes Classifier in R.

# Loading the dataset
data("Sonar")

This dataset contains 208 rows of data with 61 variables collected by reflecting sonar waves off a metal cylinder and a rock at various angles and situations. We will use the head() function to quickly print the first few rows of the dataset.
# printing the first few rows of the dataset
head(Sonar)

Output:
```
> head(Sonar)
      V1     V2     V3     V4     V5     V6     V7     V8     V9    V10    V11
1 0.0200 0.0371 0.0428 0.0207 0.0954 0.0986 0.1539 0.1601 0.3109 0.2111 0.1609
2 0.0453 0.0523 0.0843 0.0689 0.1183 0.2583 0.2156 0.3481 0.3337 0.2872 0.4918
3 0.0262 0.0582 0.1099 0.1083 0.0974 0.2280 0.2431 0.3771 0.5598 0.6194 0.6333
4 0.0100 0.0171 0.0623 0.0205 0.0205 0.0368 0.1098 0.1276 0.0598 0.1264 0.0881
5 0.0762 0.0666 0.0481 0.0394 0.0590 0.0649 0.1209 0.2467 0.3564 0.4459 0.4152
6 0.0286 0.0453 0.0277 0.0174 0.0384 0.0990 0.1201 0.1833 0.2105 0.3039 0.2988
     V12    V13    V14    V15    V16    V17    V18    V19    V20    V21    V22
1 0.1582 0.2238 0.0645 0.0660 0.2273 0.3100 0.2999 0.5078 0.4797 0.5783 0.5071
2 0.6552 0.6919 0.7797 0.7464 0.9444 1.0000 0.8874 0.8024 0.7818 0.5212 0.4052
3 0.7060 0.5544 0.5320 0.6479 0.6931 0.6759 0.7551 0.8929 0.8619 0.7974 0.6737
4 0.1992 0.0184 0.2261 0.1729 0.2131 0.0693 0.2281 0.4060 0.3973 0.2741 0.3690
5 0.3952 0.4256 0.4135 0.4528 0.5326 0.7306 0.6193 0.2032 0.4636 0.4148 0.4292
6 0.4250 0.6343 0.8198 1.0000 0.9988 0.9508 0.9025 0.7234 0.5122 0.2074 0.3985
     V23    V24    V25    V26    V27    V28    V29    V30    V31    V32    V33
1 0.4328 0.5550 0.6711 0.6415 0.7104 0.8080 0.6791 0.3857 0.1307 0.2604 0.5121
```

```
2 0.3957 0.3914 0.3250 0.3200 0.3271 0.2767 0.4423 0.2028 0.3788 0.2947 0.1984
3 0.4293 0.3648 0.5331 0.2413 0.5070 0.8533 0.6036 0.8514 0.8512 0.5045 0.1862
4 0.5556 0.4846 0.3140 0.5334 0.5256 0.2520 0.2090 0.3559 0.6260 0.7340 0.6120
5 0.5730 0.5399 0.3161 0.2285 0.6995 1.0000 0.7262 0.4724 0.5103 0.5459 0.2881
6 0.5890 0.2872 0.2043 0.5782 0.5389 0.3750 0.3411 0.5067 0.5580 0.4778 0.3299
     V34    V35    V36    V37    V38    V39    V40    V41    V42    V43    V44
1 0.7547 0.8537 0.8507 0.6692 0.6097 0.4943 0.2744 0.0510 0.2834 0.2825 0.4256
2 0.2341 0.1306 0.4182 0.3835 0.1057 0.1840 0.1970 0.1674 0.0583 0.1401 0.1628
3 0.2709 0.4232 0.3043 0.6116 0.6756 0.5375 0.4719 0.4647 0.2587 0.2129 0.2222
4 0.3497 0.3953 0.3012 0.5408 0.8814 0.9857 0.9167 0.6121 0.5006 0.3210 0.3202
5 0.0981 0.1951 0.4181 0.4604 0.3217 0.2828 0.2430 0.1979 0.2444 0.1847 0.0841
6 0.2198 0.1407 0.2856 0.3807 0.4158 0.4054 0.3296 0.2707 0.2650 0.0723 0.1238
     V45    V46    V47    V48    V49    V50    V51    V52    V53    V54    V55
1 0.2641 0.1386 0.1051 0.1343 0.0383 0.0324 0.0232 0.0027 0.0065 0.0159 0.0072
2 0.0621 0.0203 0.0530 0.0742 0.0409 0.0061 0.0125 0.0084 0.0089 0.0048 0.0094
3 0.2111 0.0176 0.1348 0.0744 0.0130 0.0106 0.0033 0.0232 0.0166 0.0095 0.0180
4 0.4295 0.3654 0.2655 0.1576 0.0681 0.0294 0.0241 0.0121 0.0036 0.0150 0.0085
5 0.0692 0.0528 0.0357 0.0085 0.0230 0.0046 0.0156 0.0031 0.0054 0.0105 0.0110
6 0.1192 0.1089 0.0623 0.0494 0.0264 0.0081 0.0104 0.0045 0.0014 0.0038 0.0013
     V56    V57    V58    V59    V60 Class
1 0.0167 0.0180 0.0084 0.0090 0.0032     R
2 0.0191 0.0140 0.0049 0.0052 0.0044     R
3 0.0244 0.0316 0.0164 0.0095 0.0078     R
4 0.0073 0.0050 0.0044 0.0040 0.0117     R
5 0.0015 0.0072 0.0048 0.0107 0.0094     R
6 0.0089 0.0057 0.0027 0.0051 0.0062     R
```

The target variable in this dataset is Class. The Class variable represents the type of object that was detected by the sonar, either a rock or a mine.

Step 3 - Train and Test Data

To evaluate the effectiveness of our Naive Bayes classifier, we will split the dataset into two parts, i.e., a training set and a testing set. We will use the training set for model training, while the testing set will be used to measure its accuracy.

```
# Splitting the dataset
set.seed(123)  # for reproducibility
splitIndex <- createDataPartition(Sonar$Class, p = 0.7,
                                  list = FALSE,
                                  times = 1)
train_data <- Sonar[splitIndex, ]
```

```r
test_data <- Sonar[-splitIndex, ]
```

Here, we set a random seed for reproducibility. Also, we are using 70% of the data for training and 30% for testing.

Step 4 - Create a naive Bayes model
Let us now train our model using the **naiveBayes()** function with our training data.

```r
# training the model
model <- naiveBayes(Class ~ ., data = train_data)
```

Here, Class ~ . means we're using the "Class" variable as the one to predict, while all other variables in the dataset (denoted by the dot ".") are used to make that prediction.

Step 5 - Make Predictions on the Test Dataset
Next, we can make predictions on our test data using the trained model.

```r
# making predictions
predictions <- predict(model, newdata = test_data)
```

Step 6 - Check the Accuracy of the Model
Now, we will create a confusion matrix and calculate the accuracy score. The confusion matrix will tell us how many instances our model predicted correctly and incorrectly. Additionally, we will print the accuracy score, calculated by dividing the total number of occurrences in the test set by the number of properly predicted cases.

```r
# evaluating the model
confusion_matrix <- table(predictions, test_data$Class)
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(confusion_matrix)
cat("Accuracy:", round(accuracy * 100, 2), "%\n")
```

Output:

```
> print(confusion_matrix)

predictions  M  R
          M 24  4
          R  9 25
> cat("Accuracy:", round(accuracy * 100, 2), "%\n")
Accuracy: 79.03 %
```

Here our model correctly classified 24 instances as M (Mine) and 25 instances as R (Rock), while it incorrectly classified 4 instances as M (Mine) and 9 instances as R

(Rock). Therefore, the model correctly classified approximately 79.03% of all instances in our test set.

Next, let us extract a particular entry from the test dataset and make a prediction as shown in the following code:

```
# Extracting particular test dataset entry
entry <- test_data[5, ]

# Make a prediction using the Naive Bayes model
prediction <- predict(model, newdata = entry)

# Print the actual class and the predicted class
cat("Actual Class:", entry$Class, "\n")
cat("Predicted Class:", prediction, "\n")
```

Output:

```
> cat("Actual Class:", entry$Class, "\n")
Actual Class: 2
> cat("Predicted Class:", prediction, "\n")
Predicted Class: 2
```

Here, we extracted the fifth entry from the test dataset and used our trained Naive Bayes model to make a prediction for this entry. It then prints the actual Class of the entry and the predicted Class.

In conclusion, Naive Bayes in R is an efficient classification technique that has several real-world applications. Naive Bayes is an essential tool for R data analysis and modeling as it successfully identifies insights from data.
Naive Bayes may be further developed and improved to handle more complex situations, such as handling imbalanced data sets. This is possible using approaches like oversampling and undersampling.

6. k- NEAREST NEIGHBOURS(k-NN):

K-Nearest Neighbors (KNN) is a supervised machine learning model that can be used for both regression and classification tasks. The algorithm is non-parametric, which means that it doesn't make any assumption about the underlying distribution of the data. KNN is a simple and intuitive algorithm that provides good results for a wide range of classification problems. It is easy to implement and understand, and it applies to both small and large datasets. However, it comes with some drawbacks too, and the main disadvantage is that it can be computationally expensive for large datasets or high-dimensional feature spaces.

Implementation of KNN in R

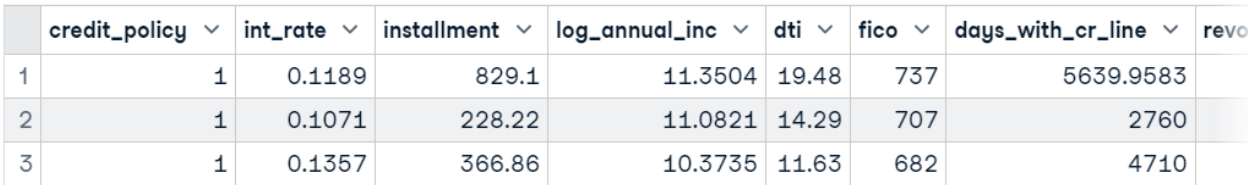
We will use Loan Data and train KNN classification using the class package. The dataset consists of 10,000 loans, and we will find whether a loan will be paid back based on the customer's data.

Step:1 Loading the Data
We will import the **tidyverse** library to access essential R packages for data loading, manipulation, and visualization. The suppressPackageStartupMessages will suppress the warnings, and you will get clean output.

After that, we will use read_csv to load the dataset, remove the "purpose" column from the dataframe using the subset function, and display the top 3 samples.

```
suppressPackageStartupMessages(library(tidyverse))

data <- read_csv('data/loans.csv.gz', show_col_types = FALSE)
data <- subset(data, select = -c(purpose))
head(data,3)
```



	credit_policy	int_rate	installment	log_annual_inc	dti	fico	days_with_cr_line	revo
1	1	0.1189	829.1	11.3504	19.48	737	5639.9583	
2	1	0.1071	228.22	11.0821	14.29	707	2760	
3	1	0.1357	366.86	10.3735	11.63	682	4710	

Step:2 Train and Test Split
We can split the dataset manually, but using the caTools library is much cleaner.
Set seed for reproducibility.
Use sample.split to create an index for training and testing datasets by a 75:25 ratio.
Use subset to create a train and test dataset, as shown below.

```
library(caTools)
set.seed(255)

split = sample.split(data$not_fully_paid,
                     SplitRatio = 0.75)
train = subset(data,
               split == TRUE)
test = subset(data,
              split == FALSE)
```

Step:3 Feature Scaling
We will now scale both the training and testing set. On the back end, the function is using (x - mean(x)) / sd(x). We are only scaling features and removing target labels from both testing and training sets.

```
train_scaled = scale(train[-13])
test_scaled = scale(test[-13])
```

Step:4 Training KNN Classifier and Predicting
The class library is quite popular for training KNN classification. It is simple and fast. We will provide a scaled train and test dataset, target column, and "k" hyperparameter.

```
library(class)
test_pred <- knn(
    train = train_scaled,
    test = test_scaled,
    cl = train$not_fully_paid,
    k=10
)
```

Step:5 Model Evaluation
To evaluate model results, we will display a confusion matrix using a table function. We have provided actual (test target) and predicted labels to the table function, and as we can see, we have quite good results for the majority class.

The KNN algorithm is not good at dealing with imbalanced data, and that is why we see poor performance in minority classes.

```
actual <- test$not_fully_paid

cm <- table(actual,test_pred)
cm
test_pred
```

```
actual    0    1
     0 1988   23
     1  373   10
```

We can calculate accuracy by summing true positive values from the confusion matrix and dividing them by the total length of target columns.

As we can observe, we have good accuracy on a vanilla model. We can improve this accuracy by tuning the “K” hyperparameter and balancing the dataset.

```
accuracy <- sum(diag(cm))/length(actual)
sprintf("Accuracy: %.2f%%", accuracy*100)
'Accuracy: 83.46%'
```

7. Support Vector Machine (SVM)

Introduction:
Support Vector Machines (SVM) are used in machine learning to classify data into two groups effectively. For our example, we will classify fruits into two categories based on their sweetness and acidity.

Mathematical Description:
SVM looks for the best hyperplane that separates the data points of two classes with the maximum margin. It uses support vectors, which are the data points nearest to the hyperplane, to define this separation most effectively.

Steps to Implement in R:

1. **Install and Load Necessary Package:**
   # To handle SVM in R, we use the e1071 package.

```
install.packages("e1071")
library(e1071)
```

2. **Prepare the Data:**
   # Here, we create a sample dataset containing sweetness and acidity measurements for two types of fruits.

```
# Creating sample data
fruits <- data.frame(
  Sweetness = c(7, 8, 6, 3, 2, 4),
  Acidity = c(5, 3, 4, 8, 7, 9),
  Type = factor(c("Apple", "Apple", "Apple", "Lemon", "Lemon", "Lemon"))
)
```

3. **Create the SVM Model:**
   We'll create an SVM model to predict the type of fruit based on Sweetness and Acidity.

```
svm_model <- svm(Type ~ ., data = fruits, type = 'C-classification', kernel = 'linear')
```

4. **Training the Model:**

The model trains on the data when we run the svm() function.

5. **Predict New Data:**
   Let's predict the type for a new fruit with given sweetness and acidity.

```
new_fruit <- data.frame(Sweetness = 5, Acidity = 6)
prediction <- predict(svm_model, new_fruit)
print(prediction)
```

6. **Input Data Example:**
   Our dataset contains values for Sweetness and Acidity for apples and lemons:

```
print(fruits)
```

7. **R Functions Used:**
   svm(): From the e1071 package to create and train the SVM model.

8. **Output Analysis:**
   We use the model to predict the type of new fruits based on their sweetness and acidity, demonstrating how SVM can classify data based on these characteristics.

```
print(prediction)
```

This output will indicate whether the new fruit is predicted to be an Apple or a Lemon, based on its sweetness and acidity levels.

8. Association Rule Mining using Apriori

Introduction:
Association Rule Mining is a technique in data mining for uncovering interesting relationships between variables in large databases. It’s famously used in market basket analysis, where you find sets of products that frequently co-occur in transactions.

Steps to Implement in R:

1. **Install and Load the Required Packages:**
   arules is an R package dedicated to association rule mining.

```
install.packages("arules")
library(arules)
```

2. **Prepare Data:**
   For simplicity, we'll simulate some transaction data.

```
transactions <- read.transactions(textConnection(
  "bread,milk
   bread,diapers,beer,eggs
   milk,diapers,beer,cola
   bread,milk,diapers,beer
   bread,milk,diapers,cola"
), format = "basket")
```

3. **Mining Association Rules:**
   We use the apriori function to find all rules with support at least 0.5 and confidence at least 0.8.

```
rules <- apriori(transactions, parameter = list(supp = 0.5, conf = 0.8))
```

4. **Viewing the Rules:**
   This step displays the rules generated by Apriori.

```
inspect(rules)
```

5. **Visualizing Rules:**

We can visualize these rules using the arulesViz package.

```
install.packages("arulesViz")
library(arulesViz)
plot(rules, method = "graph")
```

9. FP-Growth Algorithm

Introduction:
FP-Growth is a method to generate frequent item sets for association rule mining without candidate generation, which is an efficient and scalable method for mining the complete set of frequent patterns.

Steps to Implement in R:

1. **Install and Load Required Packages:**
   The arules package also supports the FP-Growth algorithm.

```
library(arules)
```

2. **Using the Same Transaction Data:**
   We can reuse the transaction data we simulated for Apriori.

```
transactions <- read.transactions(textConnection(
  "bread,milk
   bread,diapers,beer,eggs
   milk,diapers,beer,cola
   bread,milk,diapers,beer
   bread,milk,diapers,cola"
), format = "basket")
```

3. **Applying the FP-Growth Algorithm:**
   We use the fpgrowth function from the arules package.

```
fp_rules <- fpgrowth(transactions, parameter = list(supp = 0.5, conf = 0.8))
```

4. **Inspecting the Generated Rules:**
   We look at the rules generated by FP-Growth.

```
inspect(fp_rules)
```

5. **Comparing Performance:**
   Optionally, we can compare the performance and speed of Apriori vs FP-Growth for larger datasets.

Both Association Rule Mining and FP-Growth offer great insights into how items are associated within large sets of data, making them invaluable in areas like retail for cross-selling strategies.

10. K-Mean Clustering

Introduction:

K-Mean Clustering is used to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean. It's widely used for segmentation tasks, such as customer segmentation, grouping experiment outcomes, and image segmentation.

It is an unsupervised learning algorithm used for clustering, which groups data into a predefined number of clusters based on their similarities.

Steps to Implement in R:

1. **Install and Load Required Packages:**
   For K-Mean Clustering, we use the base R stats package which already includes functions for clustering.

```
# No additional packages needed for basic clustering
```

2. **Prepare the Data:**
   We'll use the iris dataset, dropping the species column to focus on clustering based on measurements alone.

```
data(iris)
iris_data <- iris[, -5]  # Remove the species column for clustering
```

3. **Perform K-Mean Clustering:**
   Here we decide to create 3 clusters, which is the same number as the species types in the iris dataset (for comparison).

```
set.seed(123)  # Set seed for reproducibility
kmeans_result <- kmeans(iris_data, centers = 3, nstart = 20)
```

4. **View the Results:**
   This includes cluster assignments and the centroids of each cluster.
   R

```
print(kmeans_result$centers)
```

5. **Plotting the Clusters:**
   Visualize the clusters by plotting them. We'll use two dimensions for simplicity.

```
library(ggplot2)
iris_data$cluster <- factor(kmeans_result$cluster)
ggplot(iris_data, aes(Sepal.Length, Sepal.Width, color = cluster)) + geom_point() +
theme_minimal()
```

6. **Analyzing Cluster Output:**
   Review the clustering results to see how well the data points have been grouped into clusters.

```
table(iris_data$cluster, iris$Species)
```

11. Rule-Based Classifier

Introduction:

A Rule-Based Classifier uses a set of "if-then" rules to make predictions. These rules are understandable and explainable, making them useful in fields where decisions need to be transparent or easily interpreted.

Steps to Implement in R:

1. **Install and Load Required Packages:**
   C5.0 is a package in R that can be used for constructing rule-based classifiers.

```
install.packages("C50")
library(C50)
```

2. **Prepare the Data:**
   We'll continue using the iris dataset. Before applying the rule-based classifier, we ensure the target variable is a factor.

```
data(iris)
iris$Species <- as.factor(iris$Species)
```

3. **Construct the Rule-Based Model:**
   Use the C5.0 algorithm to generate a model based on rules.

```
model <- C5.0(iris[,-5], iris$Species)
```

4. **Generate Rules:**
   Extract rules from the trained model.

```
rules <- C5.0Rules(model)
print(summary(rules))
```

5. **Prediction and Evaluation:**
   Make predictions using the model and evaluate its performance.

```
predictions <- predict(rules, iris[,-5])
table(predictions, iris$Species)
```

12. Principal Component Analysis (PCA)

Introduction:
Principal Component Analysis (PCA) is a technique used to emphasize variation and bring out strong patterns in a dataset. It's often used to reduce dimensions while capturing most of the variance in the data.

It is a statistical approach that can be used to analyze high-dimensional data and capture the most important information from it. This is done by transforming the original data into a lower-dimensional space while collating highly correlated variables together. In our scenario, PCA would pick three characteristics such as monthly expense, purchase frequency, and product rating. This could make it easier to visualize and understand the data.

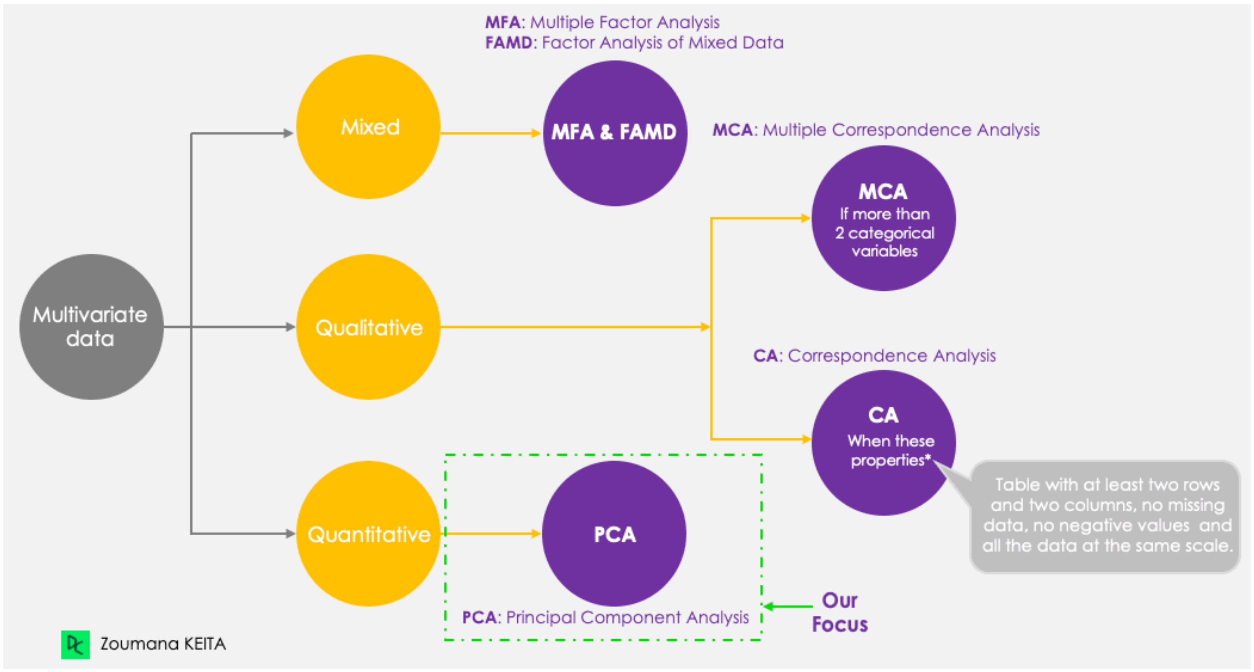

MFA: Multiple Factor Analysis
FAMD: Factor Analysis of Mixed Data
Multivariate data
Mixed
Qualitative
Quantitative
MFA & FAMD
MCA: Multiple Correspondence Analysis
MCA If more than 2 categorical variables
CA: Correspondence Analysis
CA When these properties*
Table with at least two rows and two columns, no missing data, no negative values and all the data at the same scale.
PCA
PCA: Principal Component Analysis
Our Focus
Zoumana KEITA

Fig-3 Principal component methods

Steps to Implement PCA in R:

1. **Load Required Libraries:**
   PCA can be performed using base R functions, but we'll use stats for performing PCA and ggplot2 for visualization.

```
library(stats)
library(ggplot2)
```

2. **Prepare the Data:**
   We use the iris dataset, focusing on its numeric attributes to perform PCA.

```
data(iris)
iris_data <- iris[, 1:4]  # select only numeric columns for PCA
```

3. **Perform PCA:**
   We scale the data to normalize it, which is essential for PCA because it is sensitive to variances of the initial variables.

```
pca_result <- prcomp(iris_data, center = TRUE, scale. = TRUE)
```

4. **Examine the Summary of PCA:**
   The summary provides eigenvalues and the proportion of variance explained by each principal component.

```
summary(pca_result)
```

5. **Visualize the PCA:**
   Plot the first two principal components to visualize the data reduction.

```
pca_data <- as.data.frame(pca_result$x)
pca_data$Species <- iris$Species
ggplot(pca_data, aes(PC1, PC2, color = Species)) + geom_point() + theme_minimal()
```

13. Dimensionality Reduction with t-SNE

Introduction:
t-SNE (t-Distributed Stochastic Neighbor Embedding) is a non-linear dimensionality reduction technique that is particularly well-suited for embedding high-dimensional data into a space of two or three dimensions, which can then be visualized in a scatter plot.

Steps to Implement t-SNE in R:

1. **Install and Load Required Packages:**
   The Rtsne package is specifically designed for t-SNE in R.

```
install.packages("Rtsne")
library(Rtsne)
```

2. **Prepare the Data:**
   We continue using the iris dataset but only its numeric features.

```
iris_data <- iris[, 1:4]
```

3. **Perform t-SNE:**
   The perplexity parameter is crucial; it considers a balance between the data's local and global aspects.

```
set.seed(42)  # for reproducibility
tsne_result <- Rtsne(iris_data, dims = 2, perplexity = 30)
```

4. **Plot the t-SNE Result:**
   Visualize the two-dimensional representation of the data.

```
tsne_data <- as.data.frame(tsne_result$Y)
tsne_data$Species = iris$Species
ggplot(tsne_data, aes(V1, V2, color = Species)) + geom_point() + theme_minimal()
```

These refined examples aim to provide a clearer understanding of how PCA and t-SNE can be implemented in R for dimensionality reduction, with specific emphasis on practical implementation and visualization.\