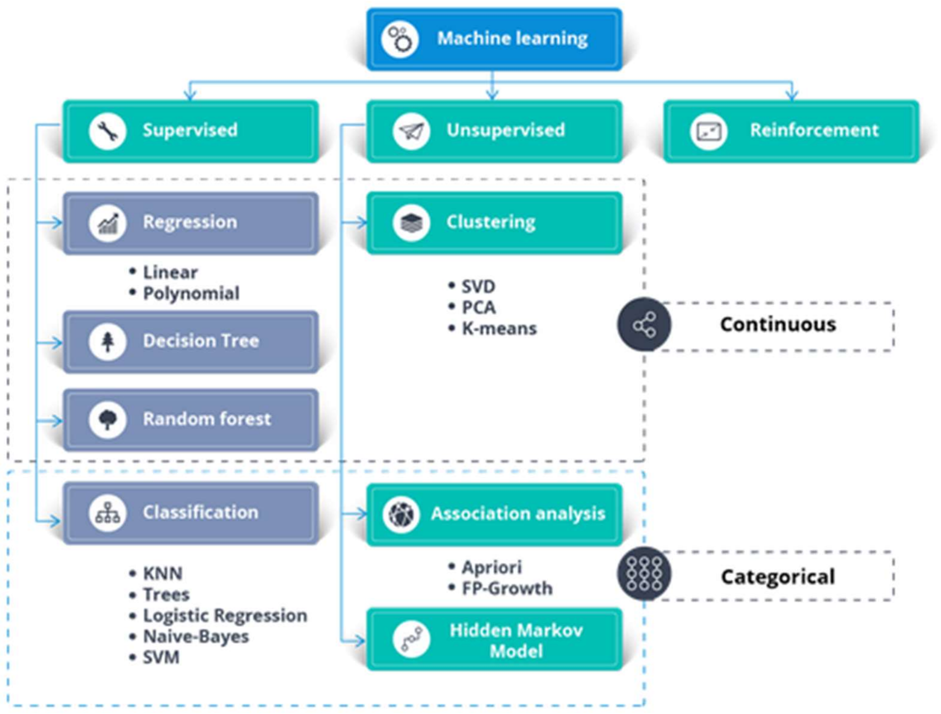
Intro to Machine Learning Algorithm



1. Linear Regression

It is used to estimate real values (cost of houses, number of calls, total sales etc.) based on continuous variables. Here, we establish a relationship between the independent and dependent variables by fitting the best line. This best fit line is known as the *regression line* and represented by a linear equation **Y= aX + b**.

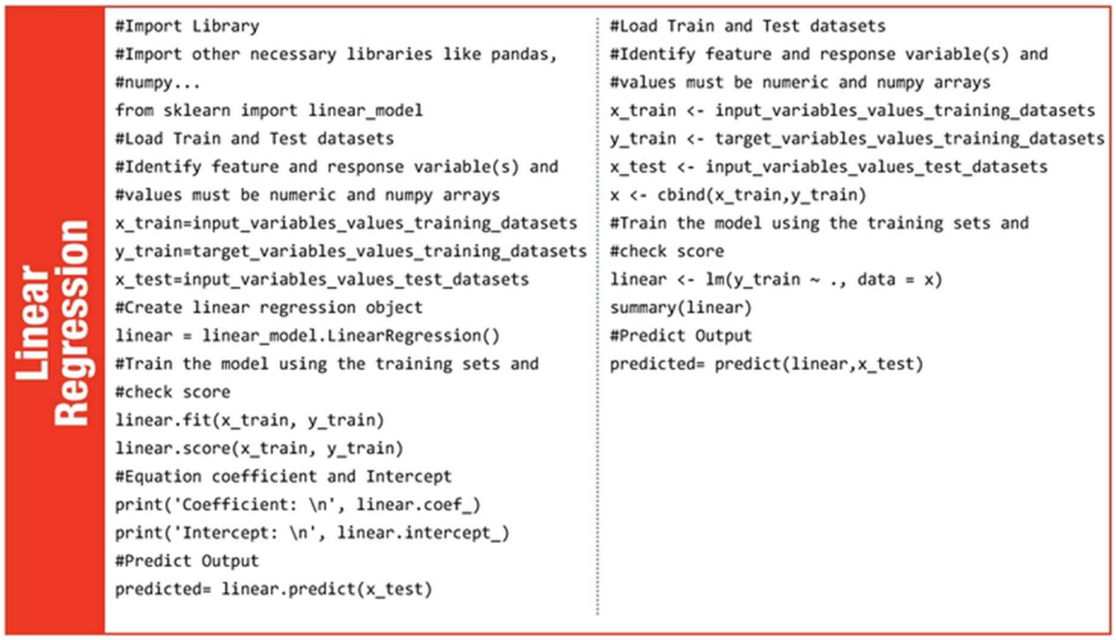
The best way to understand ***linear regression*** is to relive this experience of childhood. Let us say, you ask a child in fifth grade to arrange people in his class by increasing order of weight, without asking them their weights! What do you think the child will do? He/she would likely look (visually analyze) at the height and build of people and arrange them using a combination of these visible parameters. This is a linear regression in real life! The child has actually figured out that height and build would be correlated to the weight by a relationship, which looks like the equation above.

In this equation:

- **Y – Dependent Variable**
- **a – Slope**
- **X – Independent variable**
- **b – Intercept**

- These coefficients **a** and **b** are derived based on minimizing the 'sum of squared differences' of distance between data points and regression line.
- Look at the plot given. Here, we have identified the best fit having linear equation **y=0.2811x+13.9**. Now using this equation, we can find the weight, knowing the height of a person.

Implementation



```
#Import Library
#Import other necessary libraries like pandas,
#numpy...
from sklearn import linear_model
#Load Train and Test datasets
#Identify feature and response variable(s) and
#values must be numeric and numpy arrays
x_train=input_variables_values_training_datasets
y_train=target_variables_values_training_datasets
x_test=input_variables_values_test_datasets
#Create linear regression object
linear = linear_model.LinearRegression()
#Train the model using the training sets and
#check score
linear.fit(x_train, y_train)
linear.score(x_train, y_train)
#Equation coefficient and Intercept
print('Coefficient: \n', linear.coef_)
print('Intercept: \n', linear.intercept_)
#Predict Output
predicted= linear.predict(x_test)
```

```
#Load Train and Test datasets
#Identify feature and response variable(s) and
#values must be numeric and numpy arrays
x_train <- input_variables_values_training_datasets
y_train <- target_variables_values_training_datasets
x_test <- input_variables_values_test_datasets
x <- cbind(x_train,y_train)
#Train the model using the training sets and
#check score
linear <- lm(y_train ~ ., data = x)
summary(linear)
#Predict Output
predicted= predict(linear,x_test)
```

2. Logistic Regression

Don't get confused by its name! It is a classification, and not a regression algorithm. It is used to estimate discrete values (Binary values like 0/1, yes/no, true/false) based on a given set of independent variables(s). In simple words, it predicts the probability of occurrence of an event by fitting data to a *logit function*. Hence, it is also known as **logit regression**. Since it predicts the probability, its output values lie between 0 and 1.

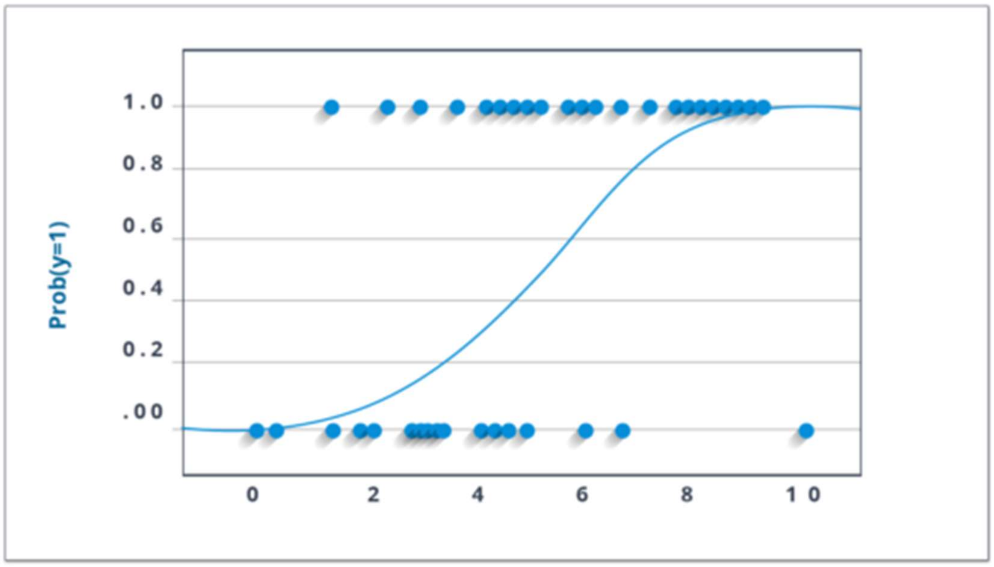
Again, let us try and understand this through a simple example.

Let's say your friend gives you a puzzle to solve. There are only 2 outcome scenarios – either you solve it or you don't. Now imagine, that you are being given a wide range of puzzles/quizzes in an attempt to understand which subjects you are good at. The outcome of this study would be something like this – if you are given a trigonometry based tenth-grade problem, you are 70% likely to solve it. On the other hand, if it is grade fifth history question, the probability of getting an answer is only 30%. This is what Logistic Regression provides you.

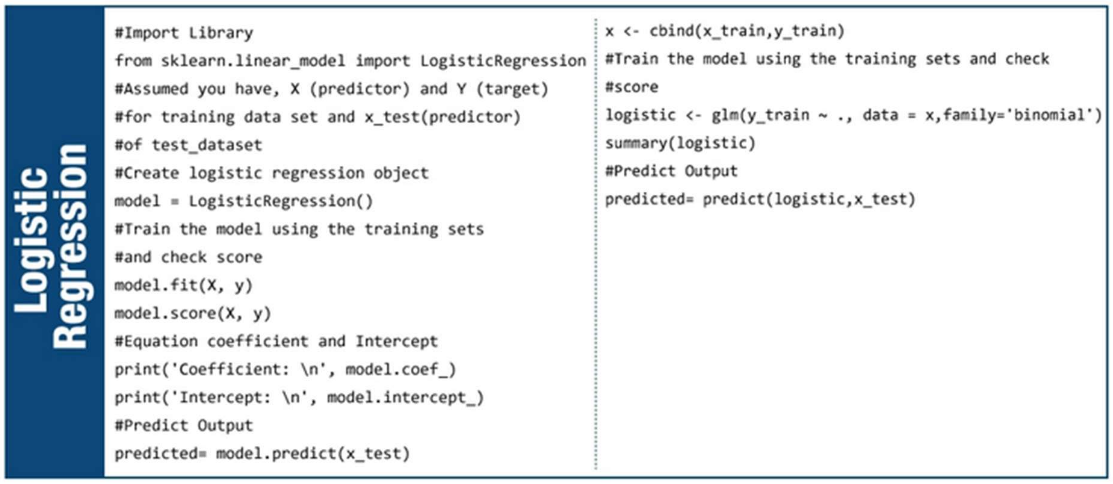
Coming to the math, the log odds of the outcome is modeled as a linear combination of the predictor variables.

odds= p/ (1-p) = probability of event occurrence / probability of not event occurrence ln(odds) = ln(p/(1-p)) logit(p) = ln(p/(1-p)) = b0+b1X1+b2X2+b3X3.... +bkX

Above, *p* is the probability of the presence of the characteristic of interest. It chooses parameters that maximize the likelihood of observing the sample values rather than that minimize the sum of squared errors (like in ordinary regression).



Implementation

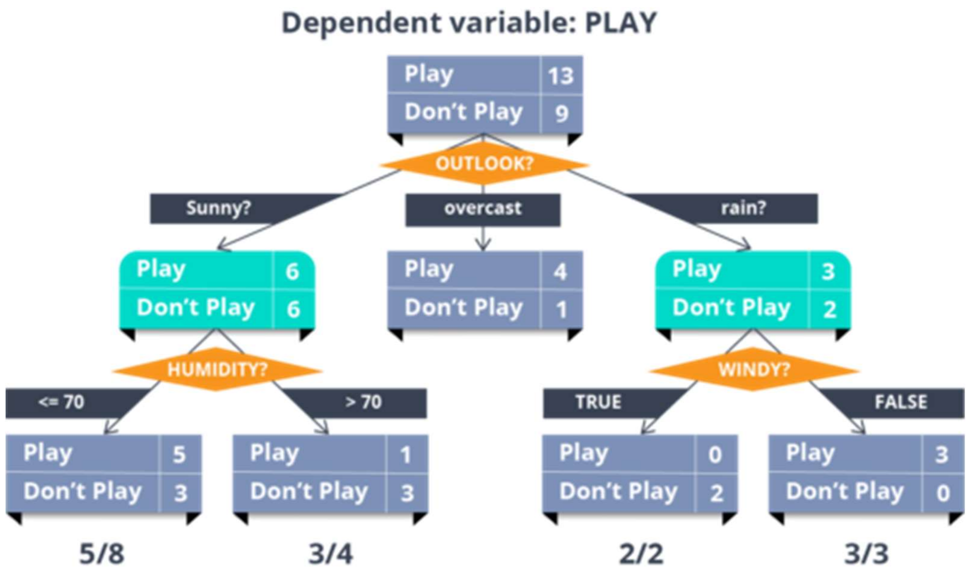


```
#Import Library
from sklearn.linear_model import LogisticRegression
#Assumed you have, X (predictor) and Y (target)
#for training data set and x_test(predictor)
#of test_dataset
#Create logistic regression object
model = LogisticRegression()
#Train the model using the training sets
#and check score
model.fit(X, y)
model.score(X, y)
#Equation coefficient and Intercept
print('Coefficient: \n', model.coef_)
print('Intercept: \n', model.intercept_)
#Predict Output
predicted= model.predict(x_test)
```

```
x <- cbind(x_train,y_train)
#Train the model using the training sets and check
#score
logistic <- glm(y_train ~ ., data = x,family='binomial')
summary(logistic)
#Predict Output
predicted= predict(logistic,x_test)
```

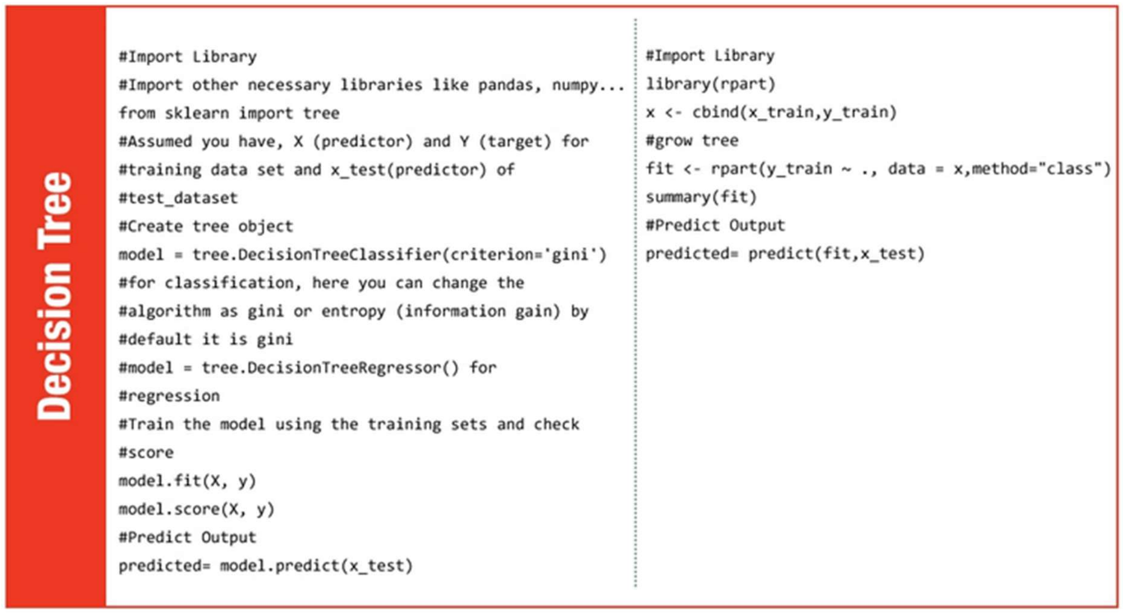
3. Decision Tree

Now, this is one of my favorite algorithms. It is a type of supervised learning algorithm that is mostly used for classification problems. Surprisingly, it works for both categorical and continuous dependent variables. In this algorithm, we split the population into two or more homogeneous sets. This is done based on the most significant attributes/ independent variables to make as distinct groups as possible.



In the image above, you can see that population is classified into four different groups based on multiple attributes to identify 'if they will play or not'.

Implementation


Decision Tree

```
#Import Library
#Import other necessary libraries like pandas, numpy...
from sklearn import tree
#Assumed you have, X (predictor) and Y (target) for
#training data set and x_test(predictor) of
#test_dataset
#Create tree object
model = tree.DecisionTreeClassifier(criterion='gini')
#for classification, here you can change the
#algorithm as gini or entropy (information gain) by
#default it is gini
#model = tree.DecisionTreeRegressor() for
#regression
#Train the model using the training sets and check
#score
model.fit(X, y)
model.score(X, y)
#Predict Output
predicted= model.predict(x_test)
```

```
#Import Library
library(rpart)
x <- cbind(x_train,y_train)
#grow tree
fit <- rpart(y_train ~ ., data = x,method="class")
summary(fit)
#Predict Output
predicted= predict(fit,x_test)
```

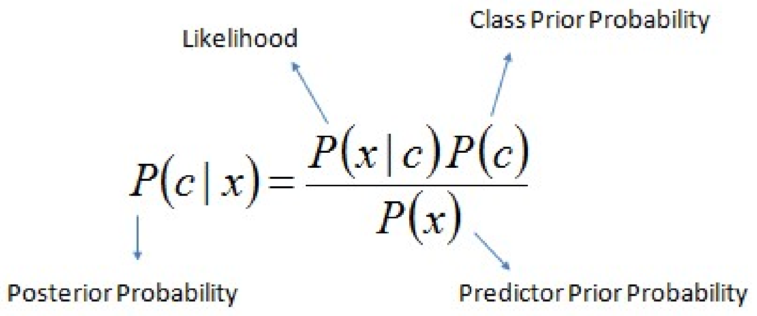
4. Naive Bayes

This is a classification technique based on *Bayes' theorem* with an assumption of independence between predictors. In simple terms, a ***Naive Bayes classifier*** assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, a naive Bayes classifier would consider all of these properties to independently contribute to the probability that this fruit is an apple.

Naive Bayesian model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

Bayes theorem provides a way of calculating posterior probability **P(c|x)** from **P(c)**, **P(x)** and **P(x|c)**. Look at the equation below:



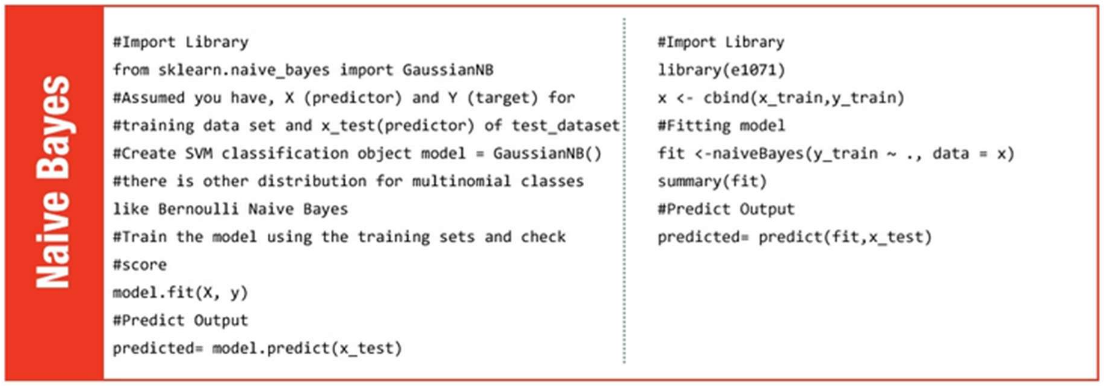
$$P(c \mid x) = \frac{P(x \mid c)P(c)}{P(x)}$$

$$P(c \mid \mathrm{X}) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

Here,

- ***P*(*c*|*x*) is the posterior probability of *class* (*target*) given *predictor* (*attribute*).**
- ***P*(*c*) is the prior probability of *class*.**
- ***P*(*x*|*c*) is the likelihood which is the probability of *predictor* given *class*.**
- ***P*(*x*) is the prior probability of *predictor*.**

Implementation:



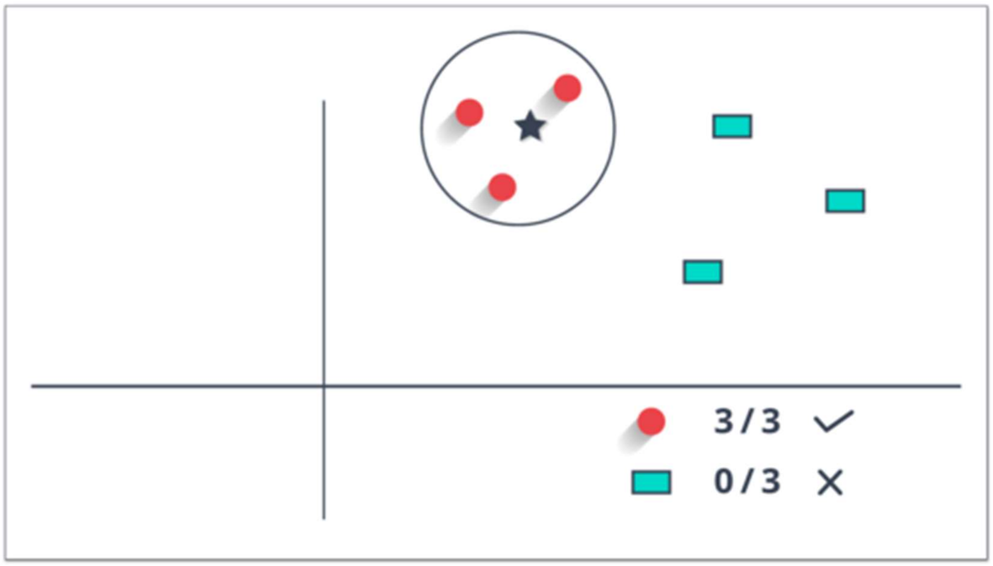
```
#Import Library
from sklearn.naive_bayes import GaussianNB
#Assumed you have, X (predictor) and Y (target) for
#training data set and x_test(predictor) of test_dataset
#Create SVM classification object model = GaussianNB()
#there is other distribution for multinomial classes
like Bernoulli Naive Bayes
#Train the model using the training sets and check
#score
model.fit(X, y)
#Predict Output
predicted= model.predict(x_test)
```

```
#Import Library
library(e1071)
x <- cbind(x_train,y_train)
#Fitting model
fit <-naiveBayes(y_train ~ ., data = x)
summary(fit)
#Predict Output
predicted= predict(fit,x_test)
```

kNN (k- Nearest Neighbors)

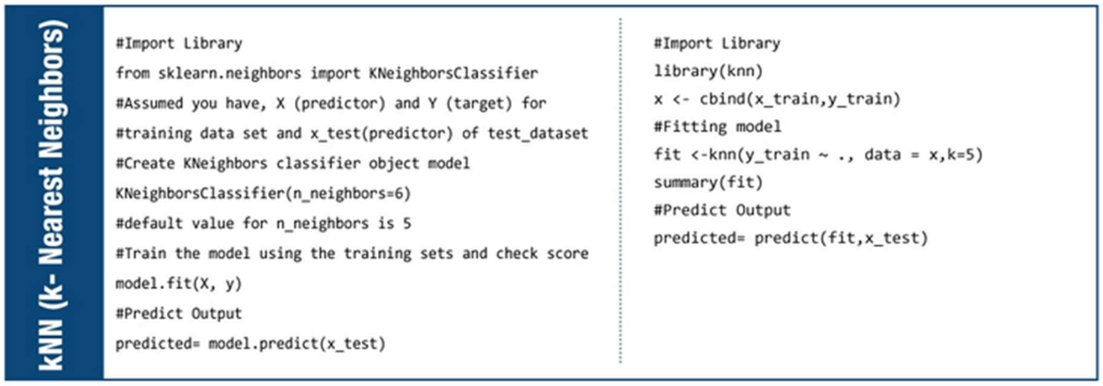
It can be used for both classification and regression problems. However, it is more widely used in classification problems in the industry. ***K nearest neighbors*** is a simple algorithm that stores all available cases and classifies new cases by a majority vote of its k neighbors. The case being assigned to the class is most common amongst its K nearest neighbors measured by a distance function.

These distance functions can be Euclidean, Manhattan, Minkowski and Hamming distance. First three functions are used for continuous function and the fourth one (Hamming) for categorical variables. If **K = 1**, then the case is simply assigned to the class of its nearest neighbor. At times, choosing K turns out to be a challenge while performing kNN modeling.



KNN can easily be mapped to our real lives. If you want to learn about a person, of whom you have no information, you might like to find out about his close friends and the circles he moves in and gain access to his/her information!

Implementation



```
#Import Library
from sklearn.neighbors import KNeighborsClassifier
#Assumed you have, X (predictor) and Y (target) for
#training data set and x_test(predictor) of test_dataset
#Create KNeighbors classifier object model
KNeighborsClassifier(n_neighbors=6)
#default value for n_neighbors is 5
#Train the model using the training sets and check score
model.fit(X, y)
#Predict Output
predicted= model.predict(x_test)
```

```
#Import Library
library(knn)
x <- cbind(x_train,y_train)
#Fitting model
fit <-knn(y_train ~ ., data = x,k=5)
summary(fit)
#Predict Output
predicted= predict(fit,x_test)
```