Commonly used Machine Learning Algorithms (R Codes)

Overview

- Major focus on commonly used machine learning algorithms
- Algorithms covered- Linear regression, logistic regression, Naive Bayes, kNN, Random Forest, etc.
- Learn both theory and implementation of these algorithms in R

Introduction

Google's self-driving cars and robots get a lot of press, but the company's real future is in machine learning, the technology that enables computers to get smarter and more personal.

– Eric Schmidt (Google Chairman)

We are probably living in the most defining period of human history. The period when computing moved from large mainframes to PCs to the cloud. But what makes it defining is not what has happened, but what is coming our way in years to come.

What makes this period exciting and enthralling for someone like me is the democratization of the various tools and techniques, which followed the boost in computing. Welcome to the world of Data Science!

Broadly, there are 3 Types of Machine learning Algorithms

1. Supervised Learning

How it works: This algorithm consists of a target / outcome variable (or dependent variable) which is to be predicted from a given set of predictors (independent variables). Using these set of variables, we generate a function that map inputs to desired outputs. The training process continues until the model achieves a desired level of accuracy on the training data. Examples of Supervised Learning: Regression, Decision Tree, Random Forest, KNN, Logistic Regression etc.

2. Unsupervised Learning

How it works: In this algorithm, we do not have any target or outcome variable to predict / estimate. It is used for clustering population in different groups, which is widely used for segmenting customers in different groups for specific intervention. Examples of Unsupervised Learning: Apriori algorithm, K-means.

3. Reinforcement Learning:

How it works: Using this algorithm, the machine is trained to make specific decisions. It works this way: the machine is exposed to an environment where it trains itself continually using trial and error. This machine learns from past experience and tries to capture the best possible knowledge to make accurate business decisions. Example of Reinforcement Learning: Markov Decision Process

List of Common Machine Learning Algorithms

Here is the list of commonly used machine learning algorithms. These algorithms can be applied to almost any data problem:

1. Linear Regression
2. Logistic Regression
3. Decision Tree
4. SVM
5. Naive Bayes
6. kNN
7. K-Means
8. Random Forest
9. Dimensionality Reduction Algorithms
10. Gradient Boosting algorithms
    1. GBM
    2. XGBoost
    3. LightGBM
    4. CatBoost

1. Linear Regression

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

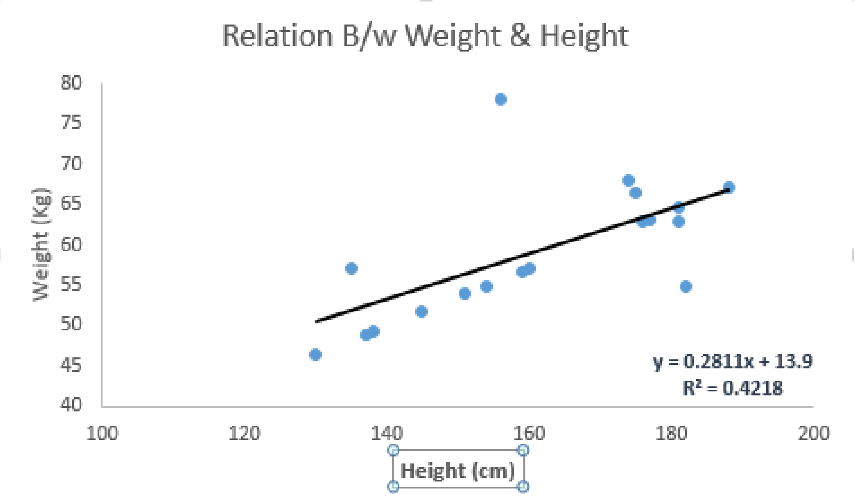
The best way to understand linear regression is to relive this experience of childhood. Let us say, you ask a child in fifth grade to arrange people in his class by increasing order of weight, without asking them their weights! What do you think the child will do? He / she would likely look (visually analyse) at the height and build of people and arrange them using a combination of these visible parameters. This is 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 difference of distance between data points and regression line.

Look at the below example. Here we have identified the best fit line having linear equation **y=0.2811x+13.9**. Now using this equation, we can find the weight, knowing the height of a person.



Linear Regression is mainly of two types: Simple Linear Regression and Multiple Linear Regression. Simple Linear Regression is characterized by one independent variable. And, Multiple Linear Regression(as the name suggests) is characterized by multiple (more than 1) independent variables. While finding the best fit line, you can fit a polynomial or curvilinear regression. And these are known as polynomial or curvilinear regression.

R Code

```
#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 not a regression algorithm. It is used to estimate discrete values (Binary values like 0/1, yes/no, true/false) based on given set of independent variable(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 lies between 0 and 1 (as expected).

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

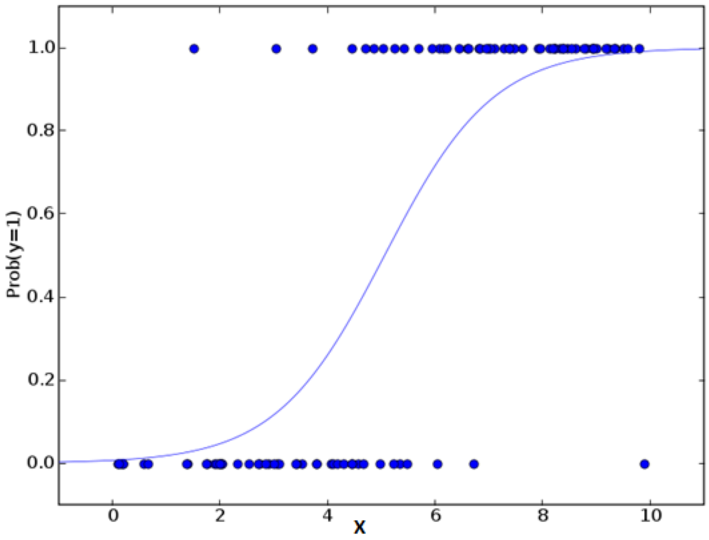
Let’s say your friend gives you a puzzle to solve. There are only 2 outcome scenarios – either you solve it or you don’t. Now imagine, that you are being given wide range of puzzles / quizzes in an attempt to understand which subjects you are good at. The outcome to this study would be something like 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 modelled 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....+bkXk
```

Above, p is the probability of 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).

Now, you may ask, why take a log? For the sake of simplicity, let’s just say that this is one of the best mathematical way to replicate a step function. I can go in more details, but that will beat the purpose of this article.



R Code

```
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)
```

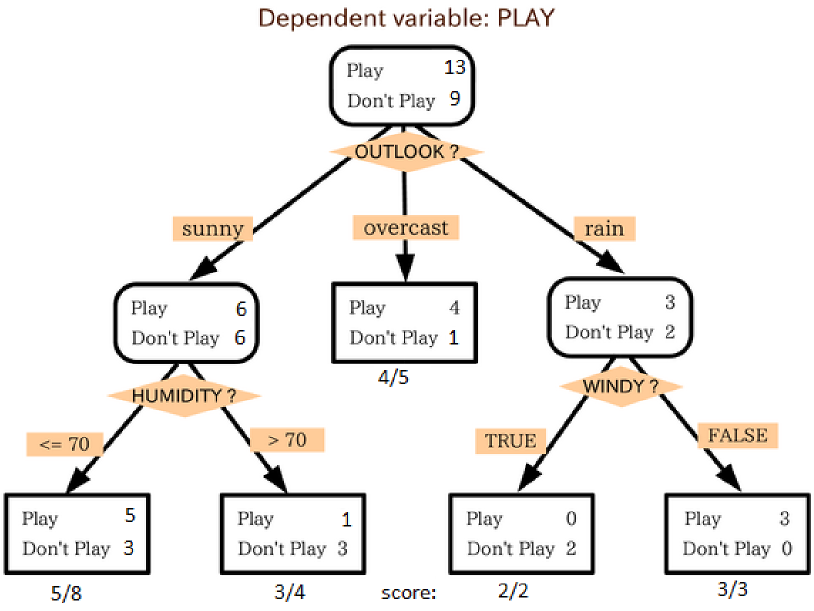
Furthermore.

There are many different steps that could be tried in order to improve the model:

- including interaction terms
- removing features
- regularization techniques
- using a non-linear model

3. Decision Tree

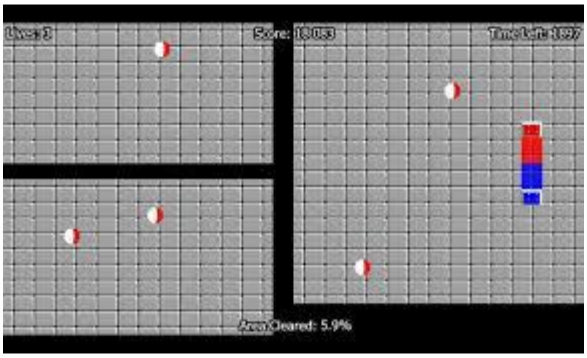
This is one of my favourite algorithms and I use it quite frequently. It is a type of supervised learning algorithm that is mostly used for classification problems. Surprisingly, it works for both categorical and continuous dependent variables. In this algorithm, we split the population into two or more homogeneous sets. This is done based on most significant attributes/ independent variables to make as distinct groups as possible. For more details, you can read: Decision Tree Simplified.



source: stats exchange

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’. To split the population into different heterogeneous groups, it uses various techniques like Gini, Information Gain, Chi-square, entropy.

The best way to understand how decision tree works, is to play Jez ball – a classic game from Microsoft (image below). Essentially, you have a room with moving walls and you need to create walls such that maximum area gets cleared off with out the balls.



So, every time you split the room with a wall, you are trying to create 2 different populations with in the same room. Decision trees work in very similar fashion by dividing a population in as different groups as possible.

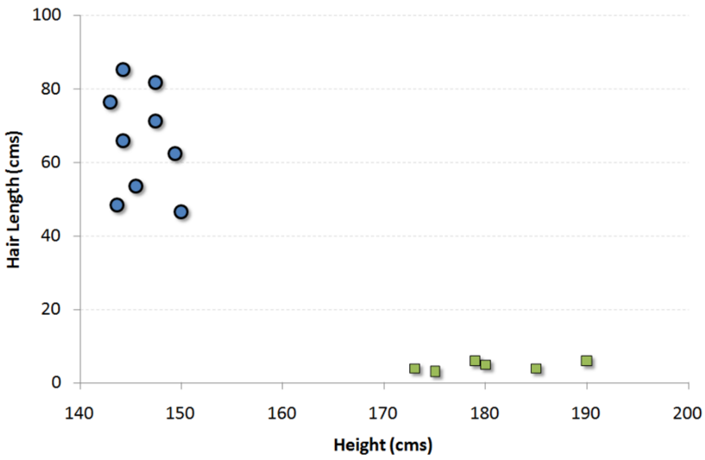
R Code

```
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)
```

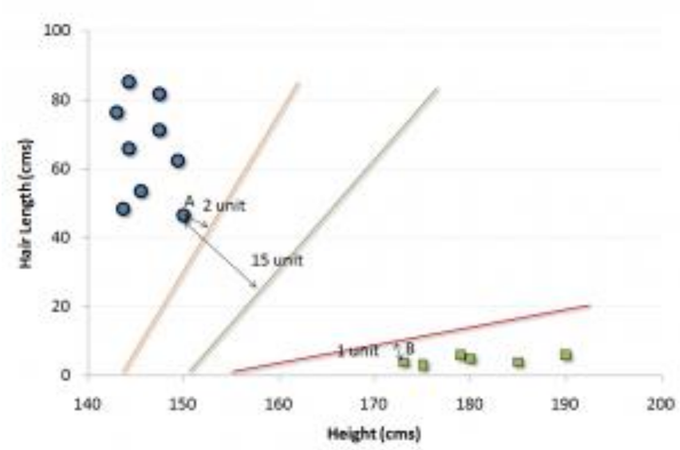
4. SVM (Support Vector Machine)

It is a classification method. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate.

For example, if we only had two features like Height and Hair length of an individual, we'd first plot these two variables in two dimensional space where each point has two co-ordinates (these co-ordinates are known as **Support Vectors**)



Now, we will find some *line* that splits the data between the two differently classified groups of data. This will be the line such that the distances from the closest point in each of the two groups will be farthest away.



In the example shown above, the line which splits the data into two differently classified groups is the *black* line, since the two closest points are the farthest apart from the line. This line is our classifier. Then, depending on where the testing data lands on either side of the line, that's what class we can classify the new data as.

More: Simplified Version of Support Vector Machine

Think of this algorithm as playing Jez Ball in n-dimensional space. The tweaks in the game are:

- You can draw lines/planes at any angles (rather than just horizontal or vertical as in the classic game)
- The objective of the game is to segregate balls of different colours in different rooms.
- And the balls are not moving.

R Code

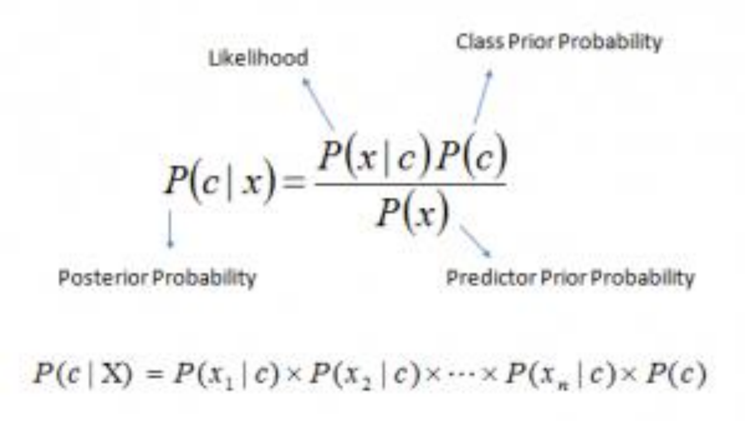
```
library(e1071)
x <- cbind(x_train,y_train)
# Fitting model
fit <-svm(y_train ~ ., data = x)
summary(fit)
#Predict Output
predicted= predict(fit,x_test)
```

5. Naive Bayes

It 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:



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

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

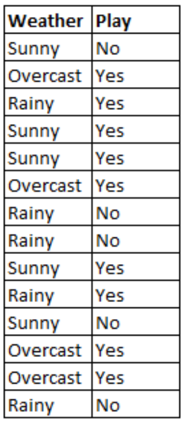
Here,

- *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*.

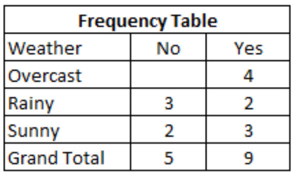
Example: Let's understand it using an example. Below I have a training data set of weather and corresponding target variable 'Play'. Now, we need to classify whether players will play or not based on weather condition. Let's follow the below steps to perform it.

Step 1: Convert the data set to frequency table

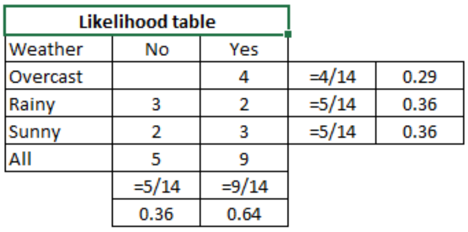
Step 2: Create Likelihood table by finding the probabilities like Overcast probability = 0.29 and probability of playing is 0.64.



Weather	Play
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No



Frequency Table		
Weather	No	Yes
Overcast		4
Rainy	3	2
Sunny	2	3
Grand Total	5	9



Likelihood table				
Weather	No	Yes		
Overcast		4	=4/14	0.29
Rainy	3	2	=5/14	0.36
Sunny	2	3	=5/14	0.36
All	5	9		
	=5/14	=9/14		
	0.36	0.64		

Step 3: Now, use Naive Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of prediction.

Problem: Players will pay if weather is sunny, is this statement is correct?

We can solve it using above discussed method, so P(Yes | Sunny) = P(Sunny | Yes) * P(Yes) / P (Sunny)

Here we have P (Sunny |Yes) = 3/9 = 0.33, P(Sunny) = 5/14 = 0.36, P(Yes)= 9/14 = 0.64

Now, P (Yes | Sunny) = 0.33 * 0.64 / 0.36 = 0.60, which has higher probability.

Naive Bayes uses a similar method to predict the probability of different class based on various attributes. This algorithm is mostly used in text classification and with problems having multiple classes.

R Code

```
library(e1071)
x <- cbind(x_train,y_train)
```

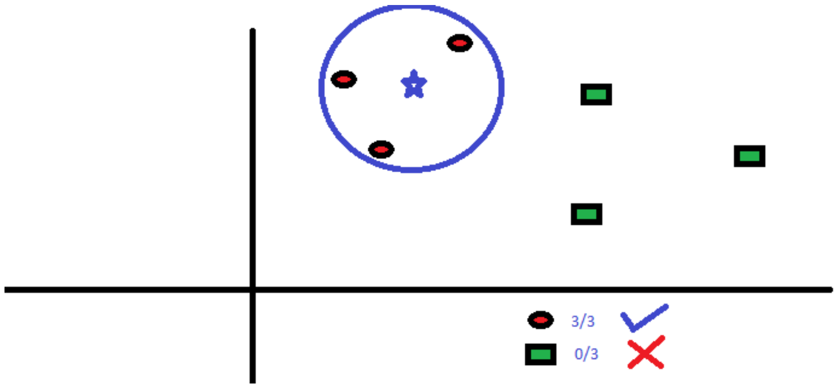
```
# Fitting model
fit <-naiveBayes(y_train ~ ., data = x)
summary(fit)
#Predict Output
predicted= predict(fit,x_test)
```

6. 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 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.

More: Introduction to k-nearest neighbors : Simplified.



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!

Things to consider before selecting kNN:

- KNN is computationally expensive
- Variables should be normalized else higher range variables can bias it
- Works on pre-processing stage more before going for kNN like an outlier, noise removal

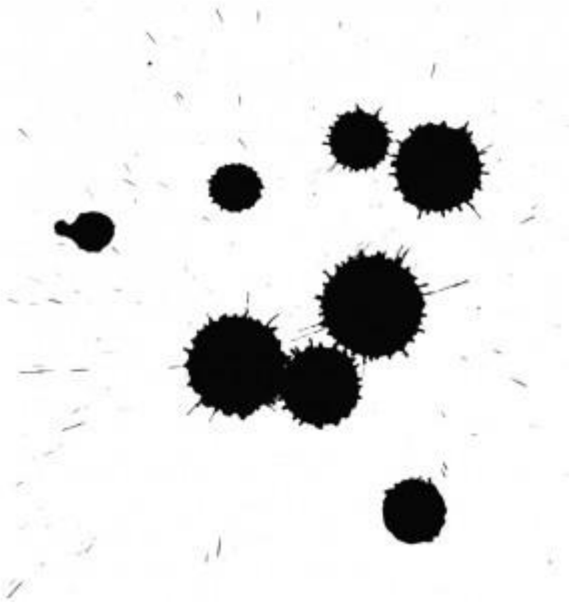
R Code

```
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)
```

7. K-Means

It is a type of unsupervised algorithm which solves the clustering problem. Its procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters). Data points inside a cluster are homogeneous and heterogeneous to peer groups.

Remember figuring out shapes from ink blots? k means is somewhat similar this activity. You look at the shape and spread to decipher how many different clusters / populations are present!



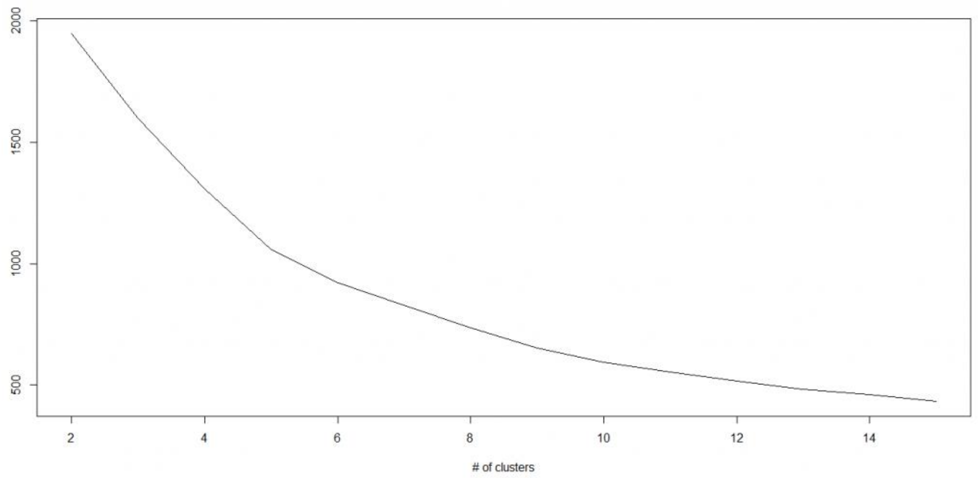
How K-means forms cluster:

1. K-means picks k number of points for each cluster known as centroids.
2. Each data point forms a cluster with the closest centroids i.e. k clusters.
3. Finds the centroid of each cluster based on existing cluster members. Here we have new centroids.
4. As we have new centroids, repeat step 2 and 3. Find the closest distance for each data point from new centroids and get associated with new k-clusters. Repeat this process until convergence occurs i.e. centroids does not change.

How to determine value of K:

In K-means, we have clusters and each cluster has its own centroid. Sum of square of difference between centroid and the data points within a cluster constitutes within sum of square value for that cluster. Also, when the sum of square values for all the clusters are added, it becomes total within sum of square value for the cluster solution.

We know that as the number of cluster increases, this value keeps on decreasing but if you plot the result you may see that the sum of squared distance decreases sharply up to some value of k, and then much more slowly after that. Here, we can find the optimum number of cluster.



R Code

```
library(cluster)
fit <- kmeans(X, 3) # 5 cluster solution
```

8. Random Forest

Random Forest is a trademark term for an ensemble of decision trees. In Random Forest, we've collection of decision trees (so known as "Forest"). To classify a new object based on attributes, each tree gives a classification and we say the tree "votes" for that class. The forest chooses the classification having the most votes (over all the trees in the forest).

Each tree is planted & grown as follows:

1. If the number of cases in the training set is N, then sample of N cases is taken at random but *with replacement*. This sample will be the training set for growing the tree.
2. If there are M input variables, a number m<<M is specified such that at each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the forest growing.
3. Each tree is grown to the largest extent possible. There is no pruning.

R Code

```
library(randomForest)

x <- cbind(x_train,y_train)

# Fitting model

fit <- randomForest(Species ~ ., x,ntree=500)

summary(fit)

#Predict Output

predicted= predict(fit,x_test)
```

9. Dimensionality Reduction Algorithms

In the last 4-5 years, there has been an exponential increase in data capturing at every possible stages. Corporates/ Government Agencies/ Research organisations are not only coming with new sources but also they are capturing data in great detail.

For example: E-commerce companies are capturing more details about customer like their demographics, web crawling history, what they like or dislike, purchase history, feedback and many others to give them personalized attention more than your nearest grocery shopkeeper.

As a data scientist, the data we are offered also consist of many features, this sounds good for building good robust model but there is a challenge. How'd you identify highly significant variable(s) out 1000 or 2000? In such cases, dimensionality reduction algorithm helps us along with various other algorithms like Decision Tree, Random Forest, PCA, Factor Analysis, Identify based on correlation matrix, missing value ratio and others.

R Code

```
library(stats)
pca <- princomp(train, cor = TRUE)
train_reduced  <- predict(pca,train)
test_reduced  <- predict(pca,test)
```

10. Gradient Boosting Algorithms

10.1. GBM

GBM is a boosting algorithm used when we deal with plenty of data to make a prediction with high prediction power. Boosting is actually an ensemble of learning algorithms which combines the prediction of several base estimators in order to improve robustness over a single estimator. It combines multiple weak or average predictors to a build strong predictor. These boosting algorithms always work well in data science competitions like Kaggle, AV Hackathon.

R Code

```
library(caret)
```

```
x <- cbind(x_train,y_train)

# Fitting model

fitControl <- trainControl( method = "repeatedcv", number = 4,
repeats = 4)

fit <- train(y ~ ., data = x, method = "gbm", trControl =
fitControl,verbose = FALSE)

predicted= predict(fit,x_test,type= "prob")[,2]
```

Gradient_Boosting_Classifier and Random Forest are two different boosting tree classifier and often people ask about the difference between these two algorithms.

10.2. XGBoost

Another classic gradient boosting algorithm that's known to be the decisive choice between winning and losing in some Kaggle competitions.

The XGBoost has an immensely high predictive power which makes it the best choice for accuracy in events as it possesses both linear model and the tree learning algorithm, making the algorithm almost 10x faster than existing gradient booster techniques.

The support includes various objective functions, including regression, classification and ranking.

One of the most interesting things about the XGBoost is that it is also called a regularized boosting technique. This helps to reduce overfit modelling and has a massive support for a range of languages such as Scala, Java, R, Python, Julia and C++.

Supports distributed and widespread training on many machines that encompass GCE, AWS, Azure and Yarn clusters. XGBoost can also be integrated with Spark, Flink and other cloud dataflow systems with a built in cross validation at each iteration of the boosting process.

R Code:

```
require(caret)


x <- cbind(x_train,y_train)


# Fitting model


TrainControl <- trainControl( method = "repeatedcv", number = 10,
repeats = 4)


model<- train(y ~ ., data = x, method = "xgbLinear", trControl =
TrainControl,verbose = FALSE)


OR


model<- train(y ~ ., data = x, method = "xgbTree", trControl =
TrainControl,verbose = FALSE)


predicted <- predict(model, x_test)
```

10.3. LightGBM

LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient with the following advantages:

- Faster training speed and higher efficiency
- Lower memory usage

- Better accuracy
- Parallel and GPU learning supported
- Capable of handling large-scale data

The framework is a fast and high-performance gradient boosting one based on decision tree algorithms, used for ranking, classification and many other machine learning tasks. It was developed under the Distributed Machine Learning Toolkit Project of Microsoft.

Since the LightGBM is based on decision tree algorithms, it splits the tree leaf wise with the best fit whereas other boosting algorithms split the tree depth wise or level wise rather than leaf-wise. So when growing on the same leaf in Light GBM, the leaf-wise algorithm can reduce more loss than the level-wise algorithm and hence results in much better accuracy which can rarely be achieved by any of the existing boosting algorithms.

R Code:

```
library(RLightGBM)

data(example.binary)

#Parameters


num_iterations <- 100

config <- list(objective = "binary",  metric="binary_logloss,auc",
learning_rate = 0.1, num_leaves = 63, tree_learner = "serial",
feature_fraction = 0.8, bagging_freq = 5, bagging_fraction = 0.8,
min_data_in_leaf = 50, min_sum_hessian_in_leaf = 5.0)


#Create data handle and booster

handle.data <- lgbm.data.create(x)


lgbm.data.setField(handle.data, "label", y)
```

```
handle.booster <- lgbm.booster.create(handle.data, lapply(config,
as.character))


#Train for num_iterations iterations and eval every 5 steps


lgbm.booster.train(handle.booster, num_iterations, 5)


#Predict
pred <- lgbm.booster.predict(handle.booster, x.test)


#Test accuracy
sum(y.test == (y.pred > 0.5)) / length(y.test)


#Save model (can be loaded again via lgbm.booster.load(filename))
lgbm.booster.save(handle.booster, filename = "/tmp/model.txt")
```

If you’re familiar with the Caret package in R, this is another way of implementing the LightGBM.

```
require(caret)
require(RLightGBM)
data(iris)


model <-caretModel.LGBM()


fit <- train(Species ~ ., data = iris, method=model, verbosity = 0)
print(fit)
y.pred <- predict(fit, iris[,1:4])


library(Matrix)
model.sparse <- caretModel.LGBM.sparse()


#Generate a sparse matrix
```

```
mat <- Matrix(as.matrix(iris[,1:4]), sparse = T)

fit <- train(data.frame(idx = 1:nrow(iris)), iris$Species, method =
model.sparse, matrix = mat, verbosity = 0)

print(fit)
```

10.4. Cat boost

CatBoost is a recently open-sourced machine learning algorithm from Yandex. It can easily integrate with deep learning frameworks like Google's TensorFlow and Apple's Core ML.

The best part about CatBoost is that it does not require extensive data training like other ML models, and can work on a variety of data formats; not undermining how robust it can be.

Make sure you handle missing data well before you proceed with the implementation.

Catboost can automatically deal with categorical variables without showing the type conversion error, which helps you to focus on tuning your model better rather than sorting out trivial errors.

R Code:

```
set.seed(1)


require(titanic)


require(caret)


require(catboost)
```

```
tt <-
titanic::titanic_train[complete.cases(titanic::titanic_train),]


data <- as.data.frame(as.matrix(tt), stringsAsFactors = TRUE)


drop_columns = c("PassengerId", "Survived", "Name", "Ticket",
"Cabin")


x <- data[,!(names(data) %in% drop_columns)]y <-
data[,c("Survived")]


fit_control <- trainControl(method = "cv", number = 4,classProbs =
TRUE)


grid <- expand.grid(depth = c(4, 6, 8),learning_rate =
0.1,iterations = 100, l2_leaf_reg = 1e-3,            rsm = 0.95,
border_count = 64)


report <- train(x, as.factor(make.names(y)),method =
catboost.caret,verbose = TRUE, preProc = NULL,tuneGrid = grid,
trControl = fit_control)


print(report)


importance <- varImp(report, scale = FALSE)


print(importance)
```