In machine learning and statistics, **classification** is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known.

Examples: assigning a given email to the "spam" or "non-spam" class.

The most commonly used include:

Linear classifiers

* Fisher's linear discriminant
* Logistic regression
* Naive Bayes classifier
* Perceptron

Support vector machines

* Least squares support vector machines

Quadratic classifiers

Kernel estimation

* k-nearest neighbor

Boosting (meta-algorithm)

Decision trees

* Random forests

Neural networks

Learning vector quantization

**Decision Trees:**

Decision tree concept is the basic classification technique. Decision trees are versatile Machine Learning algorithm that can perform both classification and regression tasks. They are very powerful algorithms, capable of fitting complex datasets. Decision tree is a graph to represent choices and their results in form of a tree.

Example:

As children, how many of you remember playing the game of twenty questions? In this game, one child would think of an animal or a place or a famous personality etc. Others would ask questions to guess it. The game would go something like this :

“I am thinking of an animal”

“Does it swim?”

“No”

“Is it carnivorous?”

“Yes”

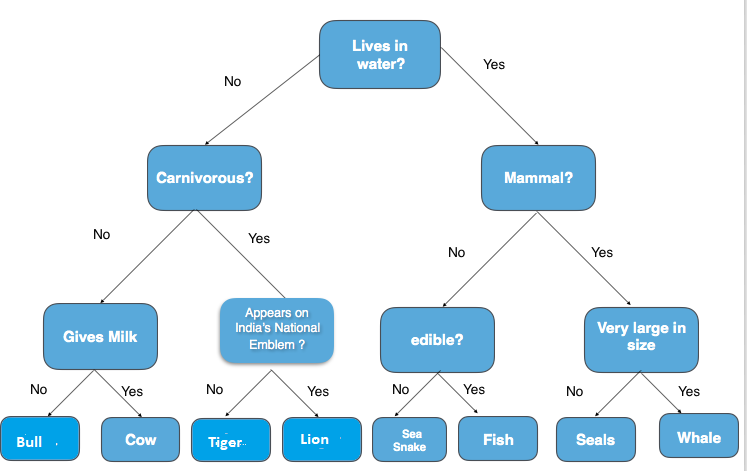
“Does it appear on the National emblem of India?”

“Yes”

“Is it a Lion?”

“Yes, it is!”

Now let’s create a little elaborate graph for “Guess the Animal “ game, we just played.



This is exactly how we would create a Decision Tree for any Data Science Problem also. Now let us study in detail the math behind it.

A Decision Tree is a supervised learning predictive model that uses a set of binary rules to calculate a target value.

* classification (categorical target variable)
* regression (continuous target variable).

Hence, it is also known as CART (Classification & Regression Trees).

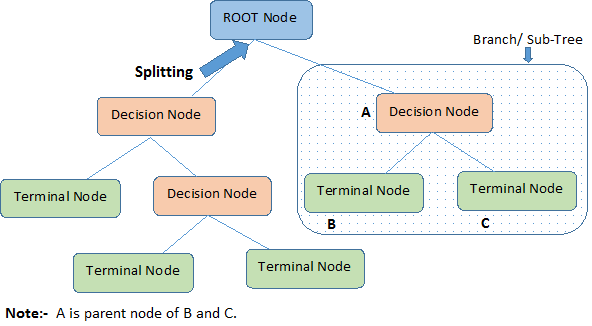
**Real-life applications of Decision Trees:**

* Credit scoring models in which the criteria that causes an applicant to be rejected need to be clearly documented and free from bias
* Marketing studies of customer behavior such as satisfaction or churn, which will be shared with management or advertising agencies
* Diagnosis of medical conditions based on laboratory measurements, symptoms, or the rate of disease progression

**Structure of a Decision Tree**

* **Root Node** represents the entire population or sample. It further gets divided into two or more homogeneous sets. In the above “Guess the Animal” example, root node would be the question lives in water.
* **Splitting** is a process of dividing a node into two or more sub-nodes.
* When a sub-node splits into further sub-nodes, it is called a **Decision Node**.
* Nodes that do not split is called a **Terminal Node** or a **Leaf**. Likewise for the example above, terminal nodes would be bull ,cow, Lion, Tiger etc
* When you remove sub-nodes of a decision node, this process is called **Pruning**. The opposite of pruning is **Splitting**.
* A sub-section of an entire tree is called **Branch**.
* A node, which is divided into sub-nodes is called a **parent node** of the sub-nodes; whereas the sub-nodes are called the **child** of the parent node.

The root node is the starting point of the tree, and both root and terminal nodes contain questions or criteria to be answered. Each node typically has two or more nodes extending from it. For example, if the question in the first node requires a “yes” or “no” answer, there will be one leaf node for a “yes” response, and another node for “no.”



**The Algorithm behind Decision Trees.**

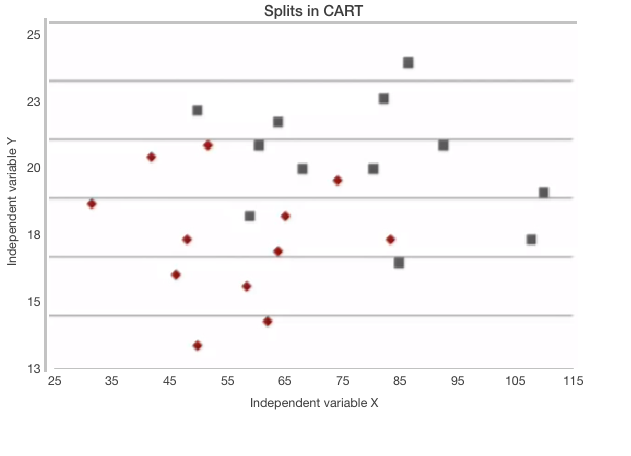
The algorithm of the decision tree models works by repeatedly partitioning the data into multiple sub-spaces, so that the outcomes in each final sub-space is as homogeneous as possible. This approach is technically called recursive partitioning.

The produced result consists of a set of rules used for predicting the outcome variable, which can be either:

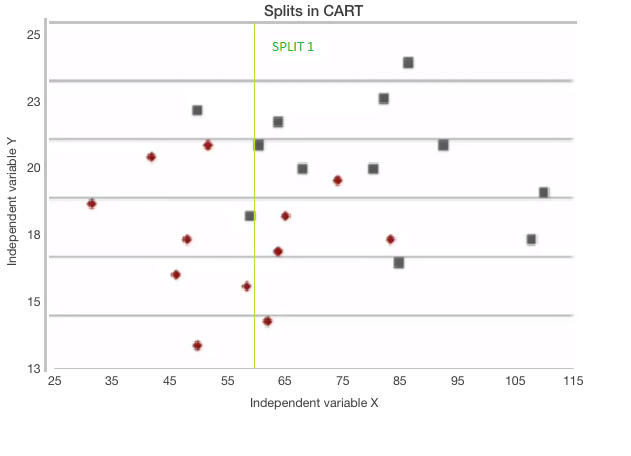
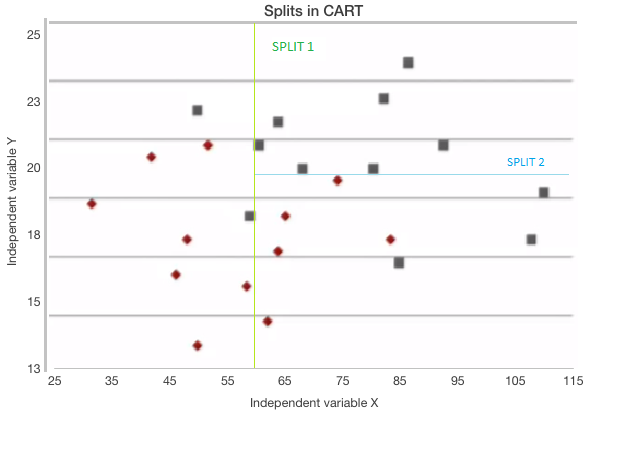
* Continuous variable, for regression trees
* Categorical variable, for classification trees

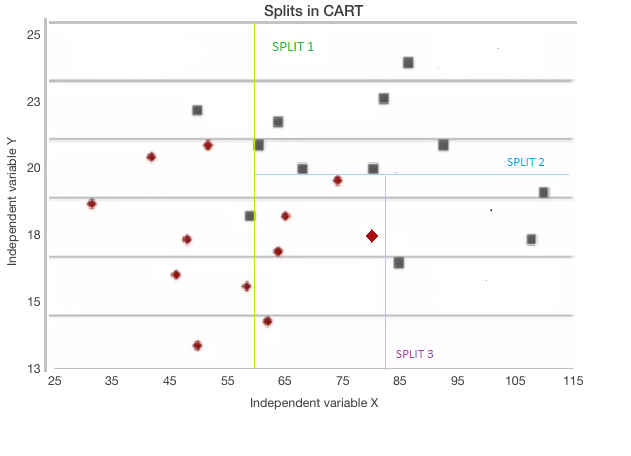
The decision rules generated by the CART (Classification & Regression Trees) predictive model are generally visualized as a binary tree.

The plot below shows a sample data for two independent variables, x and y, and each data point is colored by the outcome variable, **red or grey.**

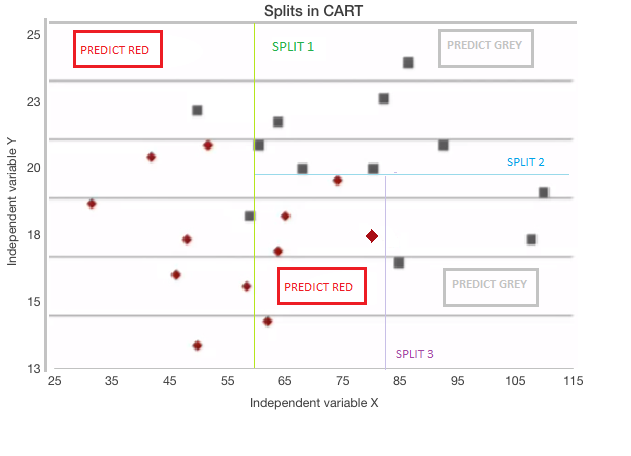


CART tries to split this data into subsets so that each subset is as pure or homogeneous as possible. The first three splits that CART would create are shown here.

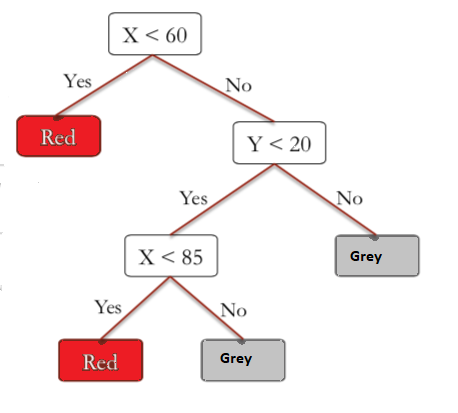
 



If a new observation fell into any of the subsets, it would now be decided by the majority of the observations in that particular subset.



The tree for the splits we just generated is shown below.



* The first split tests whether the variable x is less than 60. If yes, the model says to predict red, and if no, the model moves on to the next split.
* Then, the second split checks whether or not the variable y is less than 20. If no, the model says to predict gray, but if yes, the model moves on to the next split.
* The third split checks whether or not the variable x is less than 85. If yes, then the model says to predict red, and if no, the model says to predict grey.

**Advantages of Decision Trees**

* It is quite interpretable and easy to understand.
* It can also be used to identify the most significant variables in your dataset
* They can be displayed graphically, and they can easily handle qualitative predictors without the need to create dummy variables.
* However, decision trees generally do not have the same level of predictive accuracy as other approaches, since they aren't quite robust. A small change in the data can cause a large change in the final estimated tree.
* By aggregating many decision trees, using methods like **bagging, random forests, and boosting**, the predictive performance of decision trees can be substantially improved.

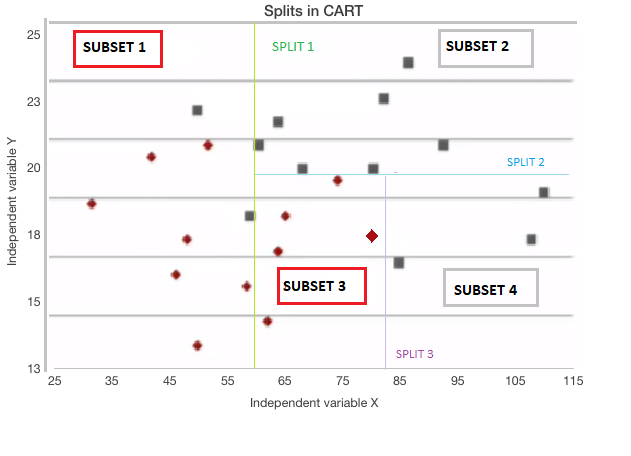
**Predictions from Decision Trees**

In the above example, we discussed Classification trees i.e when the output is a factor/category :red or gray.

Trees can also be used for regression where the output at each leaf of the tree is no longer a category, but a number. They are called Regression Trees.

Classification Trees:

With Classification Trees we report the average outcome at each leaf of our tree.

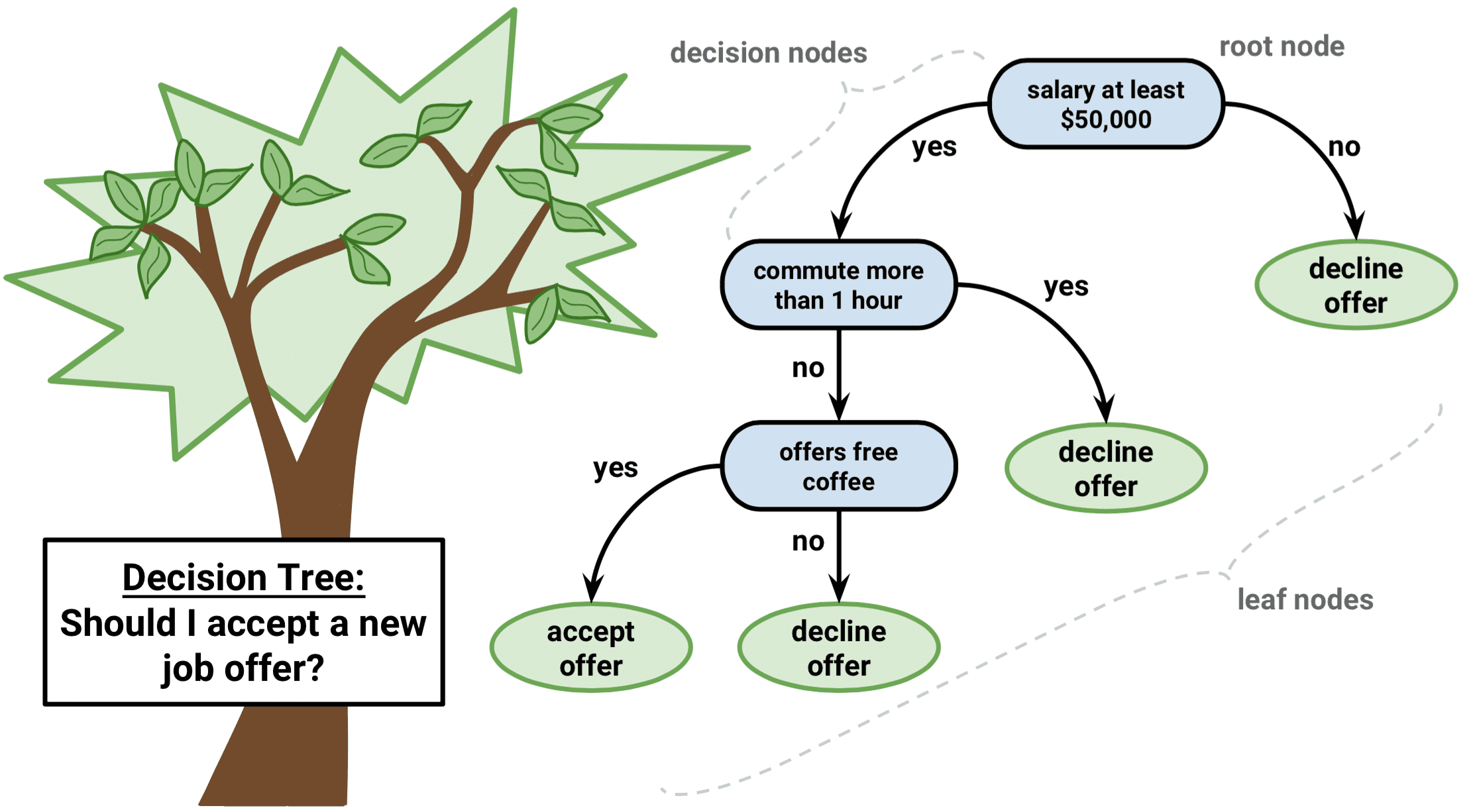


The above dataset has been split into four subsets.

**Predictions for Subset 1:**

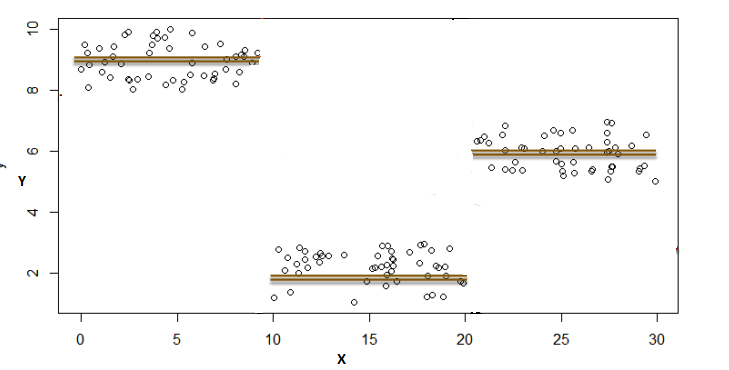
* Red data = 7, Grey data = 2
* % of Red data = 7/(7+2) ~ **78%** and % of Grey data ~**22%.**This means 78% of the data is Red.
* Now just like in Logistic Regression, we can use a threshold value to obtain our prediction.
* A Threshold of 0.5/50%, corresponds to picking most frequent outcome which would be **Red.**
* But if we increase that threshold to 0.9/90%, we would predict **Grey**

**Another Example:**



**Regression Trees:**

To predict outcome in such cases, since we have continuous output variables, we simply report the average of the values at that leaf. For example, if we had the values 3, 4, and 5 at one of the leaves, we will just take the average i.e 4.

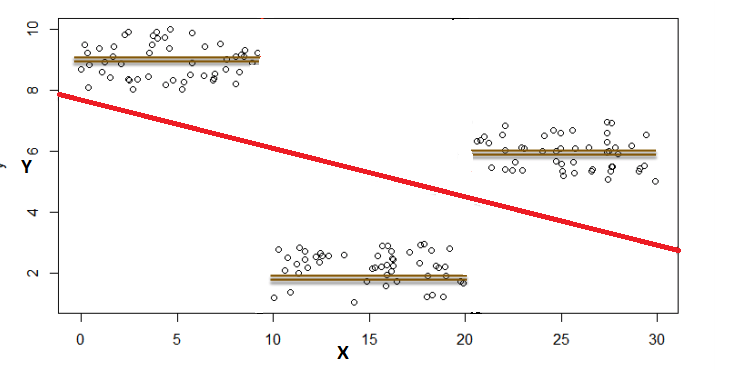


In the above graph:

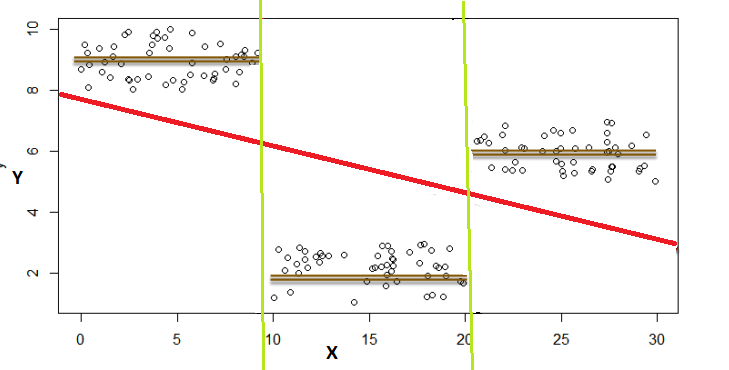
y = Outcome/target variable i.e variable we are trying to predict

x = Independent variable

Firstly, Let’s fit a linear regression to this data set . By doing so , we obtain a line .



As is quite evident, linear regression does not do very well on this data set.



However, we can notice a very interesting feature. The data lies in three different groups. If we draw lines here, we see x is either less than 10, between 10 and 20, or greater then 20.

We recall that Decision Trees can fit in this this kind of of problem easily. So if splits are at:

x ≤10 |output would be the average of those values.

10 < x ≤ 20 |output would be the average of those values.

20< x≤ 30 |output would be the average of those values.

**Measures Used for Split**

There are different ways to control how many splits are generated.

Gini Index: The Gini Index is an alternative error metric that is designed to show how "pure" a region is. "Purity" in this case means how much of the training data in a particular region belongs to a single class. If a region Rm contains data that is mostly from a single class c then the Gini Index value will be small.

Entropy: Entropy is nothing but the measure of disorder.

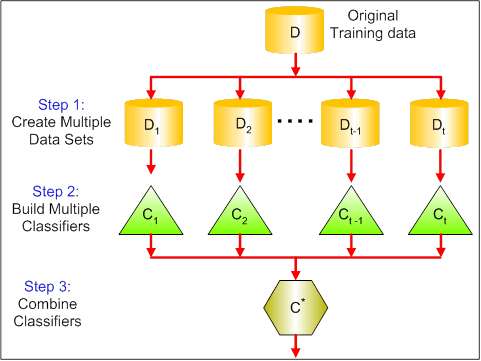
When building a classification tree, either the Gini index or the cross-entropy are typically used to evaluate the quality of a particular split, since they are more sensitive to node purity.

Information Gain: Information gain is used to decide which feature to split on at each step in building the tree. Simplicity is best, so we want to keep our tree small.

Tree-Based Methods

1. Bagging
2. Random Forest
3. Boosting

**Bagging,** or bootstrap aggregation, is a technique used to reduce the variance of your predictions by combining the result of multiple classifiers modeled on different sub-samples of the same dataset.

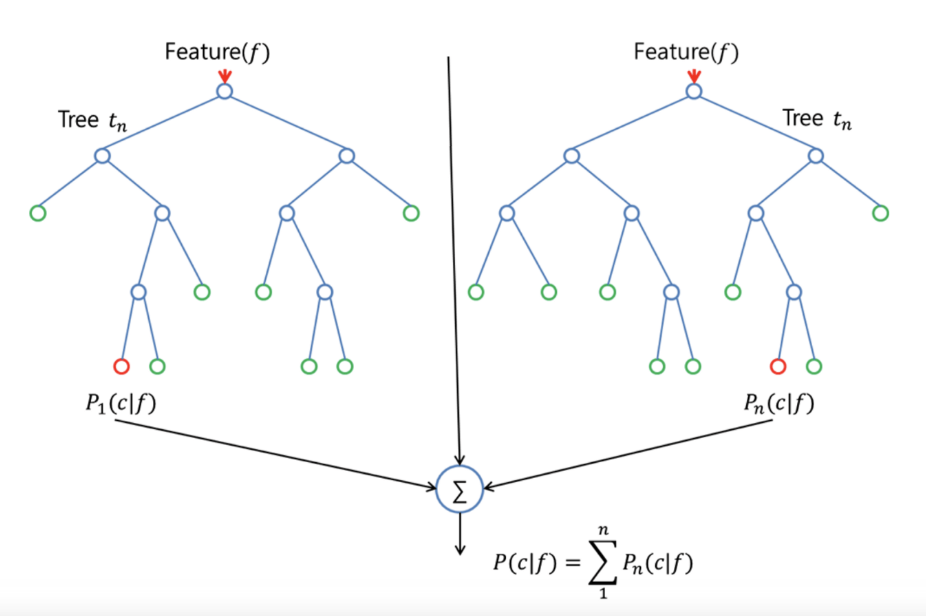


You then train your method on the bth bootstrapped training set and finally average the predictions.

Random Forests

Random Forests is a versatile machine learning method capable of performing both regression and classification tasks. It also undertakes dimensional reduction methods, treats missing values, outlier values and other essential steps of data exploration, and does a fairly good job.

Random Forests provides an improvement over bagged trees by a small tweak that decorrelates the trees. As in bagging, you build a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors. This is the main difference between random forests and bagging; because as in bagging, the choice of predictor m=p.



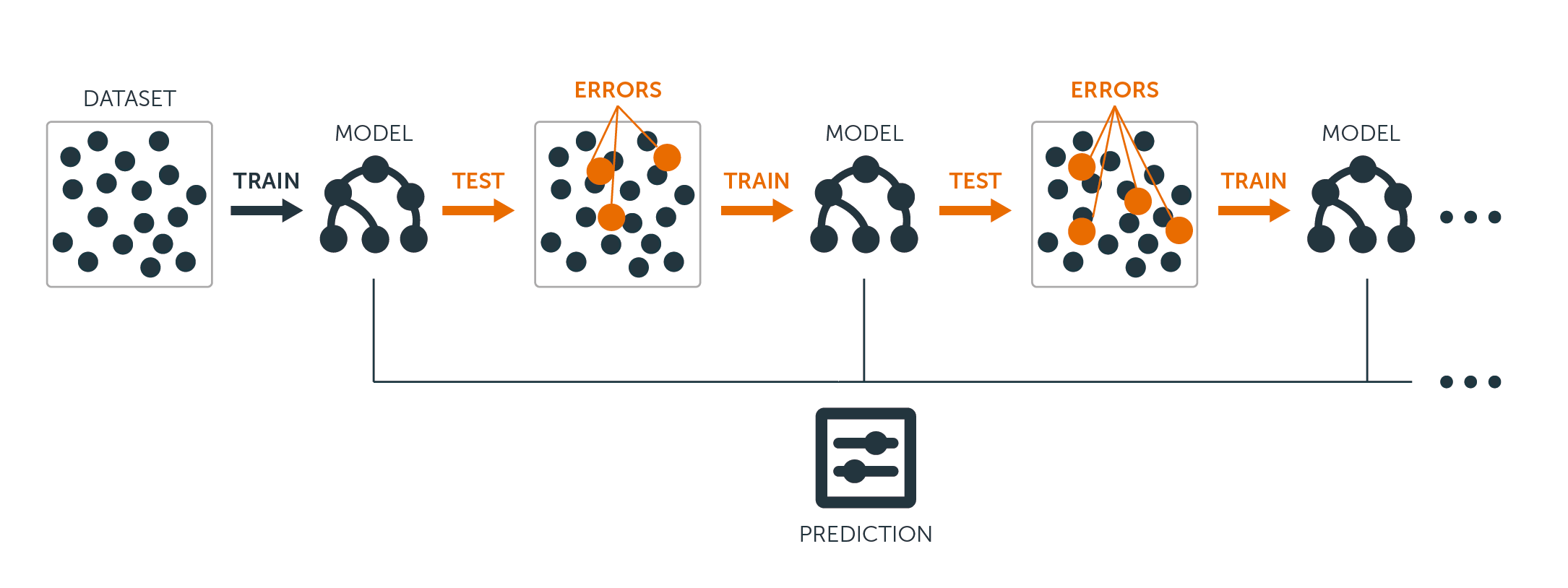
In order to grow a random forest, you should:

* First assume that the number of cases in the training set is K. Then, take a random sample of these K cases, and then use this sample as the training set for growing the tree.
* If there are p input variables, specify a number m<p such that at each node, you can select m random variables out of the p. The best split on these m is used to split the node.
* Each tree is subsequently grown to the largest extent possible and no pruning is needed.
* Finally, aggregate the predictions of the target trees to predict new data.

Random Forests is very effective at estimating missing data and maintaining accuracy when a large proportions of the data is missing. It can also balance errors in datasets where the classes are imbalanced. Most importantly, it can handle massive datasets with large dimensionality. However, one disadvantage of using Random Forests is that you might easily overfit noisy datasets, especially in the case of doing regression.

Boosting is another approach to improve the predictions resulting from a decision tree. Like bagging and random forests, it is a general approach that can be applied to many statistical learning methods for regression or classification. Recall that bagging involves creating multiple copies of the original training dataset using the bootstrap, fitting a separate decision tree to each copy, and then combining all of the trees in order to create a single predictive model. Notably, each tree is built on a bootstrapped dataset, independent of the other trees.

Boosting works in a similar way, except that the trees are grown sequentially: each tree is grown using information from previously grown trees. Boosting does not involve bootstrap sampling; instead, each tree is fitted on a modified version of the original dataset.



Boosting is very useful when you have a lot of data and you expect the decision trees to be very complex. Boosting has been used to solve many challenging classification and regression problems, including risk analysis, sentiment analysis, predictive advertising, price modeling, sales estimation and patient diagnosis, among others.

Decision Tree:

library(ISLR)

install.packages("ISLR")

library(ISLR)

data(package="ISLR")

carseats<-Carseats

require(tree)

names(carseats)

hist(carseats$Sales)

High = ifelse(carseats$Sales<=8, "No", "Yes")

carseats = data.frame(carseats, High)

head(carseats)

tree.carseats = tree(High~.-Sales, data=carseats)

summary(tree.carseats)

plot(tree.carseats)

text(tree.carseats, pretty = 0)

tree.carseats

#lets Prune the Tree

set.seed(101)

train=sample(1:nrow(carseats), 250)

tree.carseats = tree(High~.-Sales, carseats, subset=train)

plot(tree.carseats)

text(tree.carseats, pretty=0)

tree.pred = predict(tree.carseats, carseats[-train,], type="class")

with(carseats[-train,], table(tree.pred, High))

When growing a big bushy tree, it could have too much variance. Thus, let's use cross-validation to prune the tree optimally. Using cv.tree, you'll use the misclassification error as the basis for doing the pruning.

cv.carseats = cv.tree(tree.carseats, FUN = prune.misclass)

cv.carseats

plot(cv.carseats)

prune.carseats = prune.misclass(tree.carseats, best = 12)

plot(prune.carseats)

text(prune.carseats, pretty=0)

It's a bit shallower than previous trees, and you can actually read the labels. Let's evaluate it on the test dataset again.

tree.pred = predict(prune.carseats, carseats[-train,], type="class")

with(carseats[-train,], table(tree.pred, High))

Often case, trees don't give very good prediction errors, so let's go ahead take a look at random forests and boosting, which tend to outperform trees as far as prediction and misclassification are concerned.

**Random Forests**

For this part, you will use the Boston housing data to explore random forests and boosting. The dataset is located in the MASS package. It gives housing values and other statistics in each of 506 suburbs of Boston based on a 1970 census.

library(MASS)

data(package="MASS")

boston<-Boston

dim(boston)

names(boston)

require(randomForest)

set.seed(101)

train = sample(1:nrow(boston), 300)

In this dataset, there are 506 surburbs of Boston. For each surburb, you have variables such as crime per capita, types of industry, average # of rooms per dwelling, average proportion of age of the houses etc. Let's use medv - the median value of owner-occupied homes for each of these surburbs, as the response variable.

Let's fit a random forest and see how well it performs. As being said, you use the response medv, the median housing value (in $1K dollars), and the training sample set.

rf.boston = randomForest(medv~., data = boston, subset = train)

rf.boston

oob.err = double(13)

test.err = double(13)

for(mtry in 1:13){

fit = randomForest(medv~., data = boston, subset=train, mtry=mtry, ntree = 350)

oob.err[mtry] = fit$mse[350]

pred = predict(fit, boston[-train,])

test.err[mtry] = with(boston[-train,], mean( (medv-pred)^2 ))

}

Basically you just grew 4550 trees (13 times 350).

matplot(1:mtry, cbind(test.err, oob.err), pch = 23, col = c("red", "blue"), type = "b", ylab="Mean Squared Error")

legend("topright", legend = c("OOB", "Test"), pch = 23, col = c("red", "blue"))

Ideally, these 2 curves should line up, but it seems like the test error is a bit lower. However, there's a lot of variability in these test error estimates. Since the out-of-bag error estimate was computed on one dataset and the test error estimate was computed on another dataset, these differences are pretty much well within the standard errors.

Notice that the red curve is smoothly above the blue curve? These error estimates are very correlated, because the randomForest with mtry = 4 is very similar to the one with mtry = 5. That's why each of the curves is quite smooth. What you see is that mtry around 4 seems to be the most optimal choice, at least for the test error. This value of mtry for the out-of-bag error equals 9.

So with very few tiers, you have fitted a very powerful prediction model using random forests. How so? The left-hand side shows the performance of a single tree. The mean squared error on out-of-bag is 26, and you've dropped down to about 15 (just a bit above half). This means you reduced the error by half. Likewise for the test error, you reduced the error from 20 to 12.

Boosting

Compared to random forests, boosting grows smaller and stubbier trees and goes at the bias. You will use the package GBM (Gradient Boosted Modeling), in R.

require(gbm)

GBM asks for the distribution, which is Gaussian, because you'll be doing squared error loss. You're going to ask GBM for 10,000 trees, which sounds like a lot, but these are going to be shallow trees. Interaction depth is the number of splits, so you want 4 splits in each tree. Shrinkage is 0.01, which is how much you're going to shrink the tree step back.

boost.boston = gbm(medv~., data = boston[train,], distribution = "gaussian", n.trees = 10000, shrinkage = 0.01, interaction.depth = 4)

summary(boost.boston)

plot(boost.boston,i="lstat")

plot(boost.boston,i="rm")

The 1st plot shows that the higher the proportion of lower status people in the suburb, the lower the value of the housing prices. The 2nd plot shows the reversed relationship with the number of rooms: the average number of rooms in the house increases as the price increases.

It's time to predict a boosted model on the test dataset. Let's look at the test performance as a function of the number of trees:

First, you make a grid of number of trees in steps of 100 from 100 to 10,000.

Then, you run the predict function on the boosted model. It takes n.trees as an argument, and produces a matrix of predictions on the test data.

The dimensions of the matrix are 206 test observations and 100 different predict vectors at the 100 different values of tree.

n.trees = seq(from = 100, to = 10000, by = 100)

predmat = predict(boost.boston, newdata = boston[-train,], n.trees = n.trees)

dim(predmat)

It's time to compute the test error for each of the predict vectors:

predmat is a matrix, medv is a vector, thus (predmat - medv) is a matrix of differences. You can use the apply function to the columns of these square differences (the mean). That would compute the column-wise mean squared error for the predict vectors.

Then you make a plot using similar parameters to that one used for Random Forest. It would show a boosting error plot.

boost.err = with(boston[-train,], apply( (predmat - medv)^2, 2, mean) )

plot(n.trees, boost.err, pch = 23, ylab = "Mean Squared Error", xlab = "# Trees", main = "Boosting Test Error")

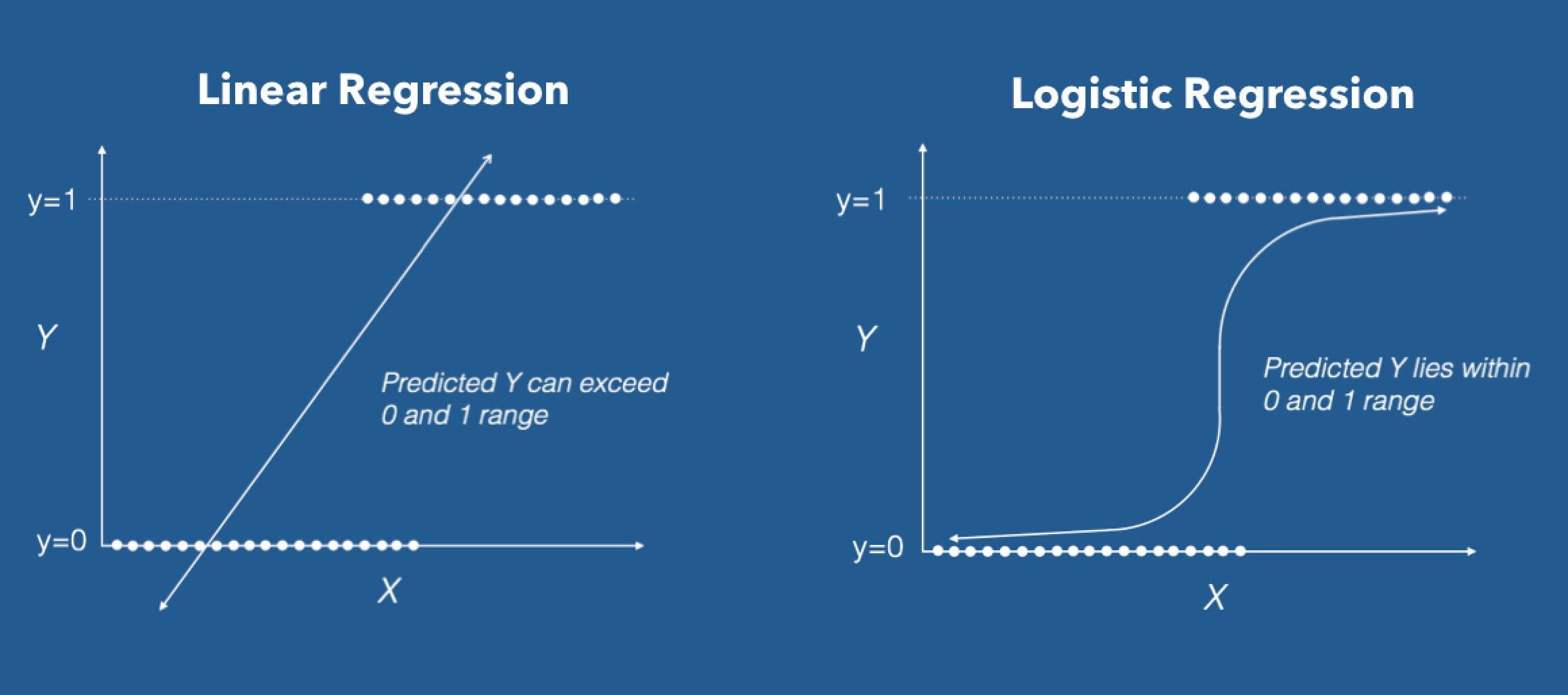
abline(h = min(test.err), col = "red")

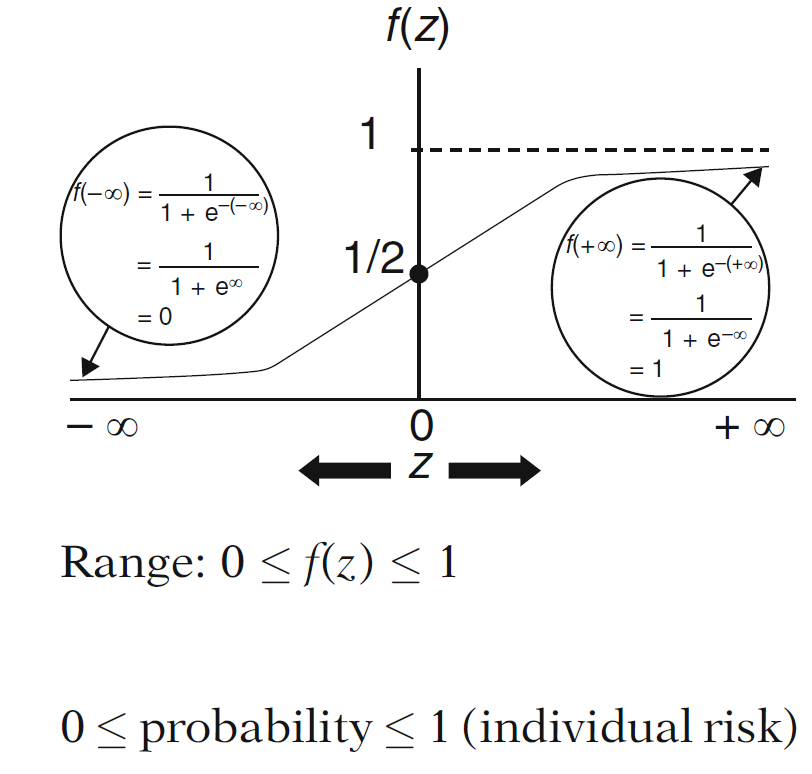
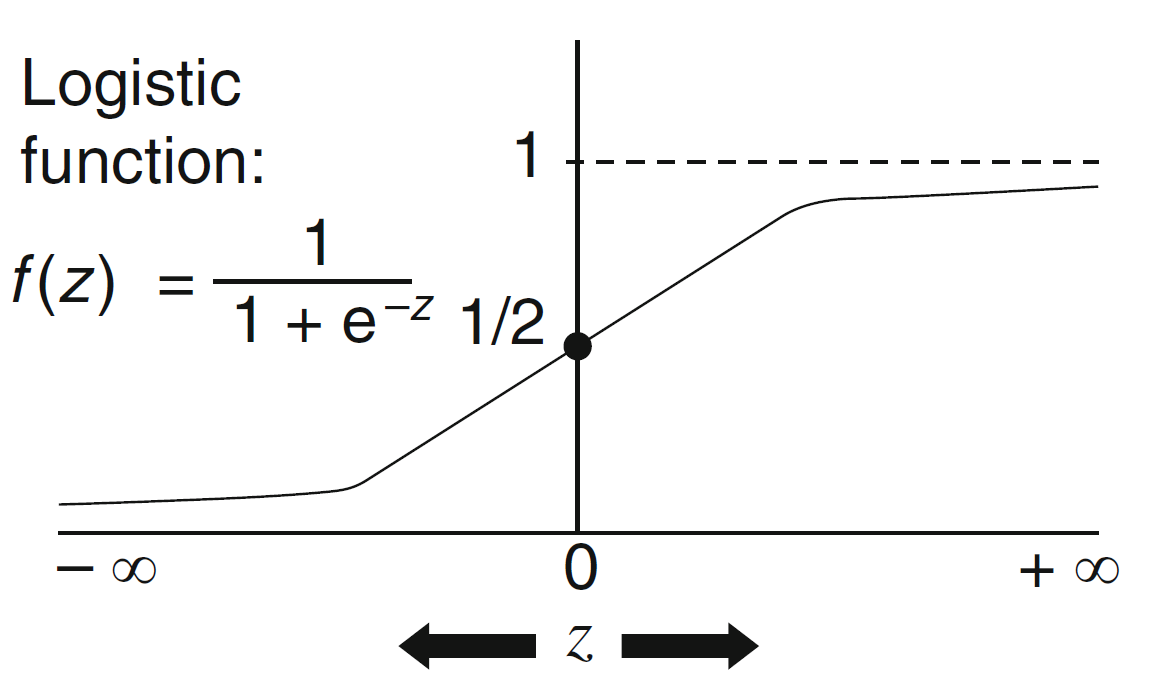
The boosting error pretty much drops down as the number of trees increases. This is an evidence showing that boosting is reluctant to overfit.

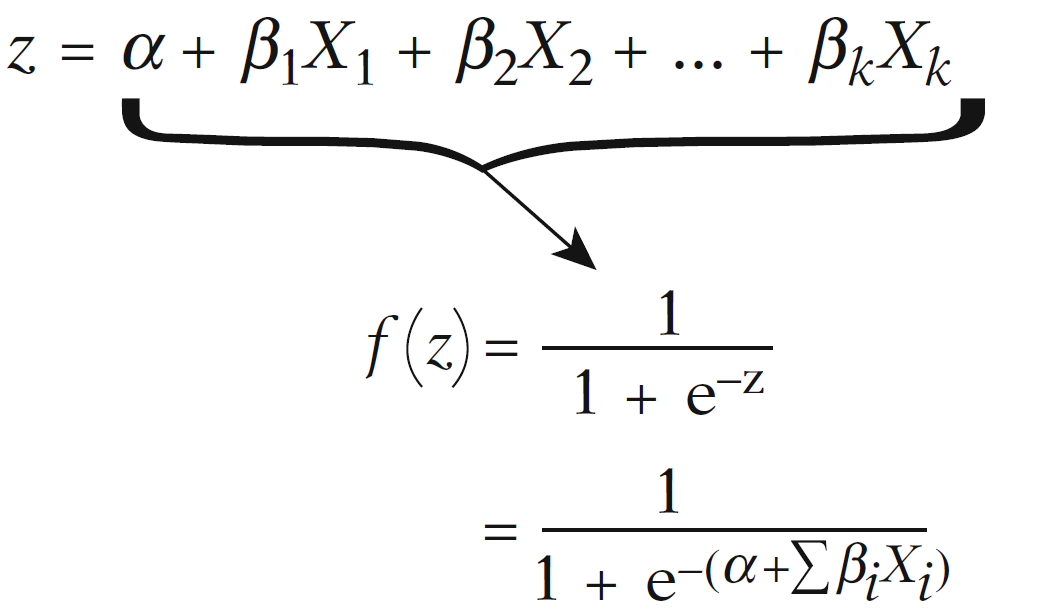
**Logistic regression**:

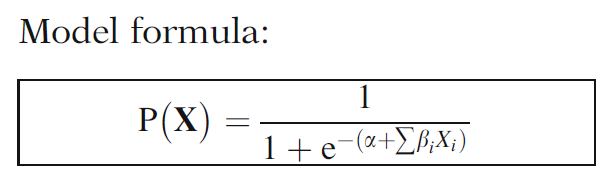
a mathematical modeling approach that can be used to describe the relationship

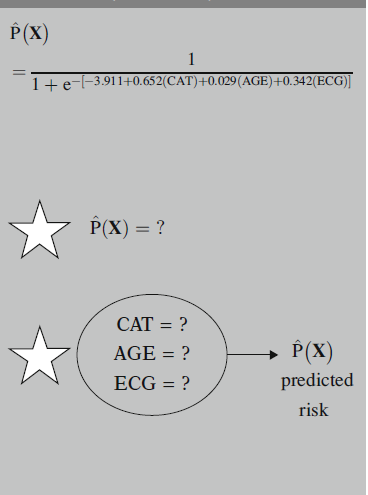
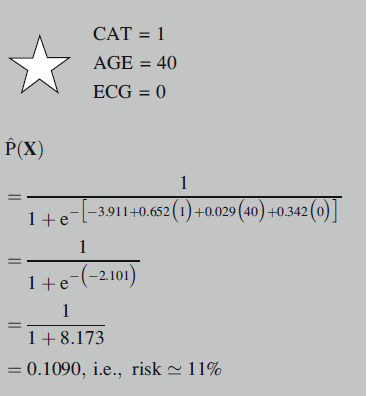
of several Xs to a dichotomous dependent variable.









The coefficients β0 and β1 are unknown, and must be estimated based on the available training data. For logistic regression, you can use maximum likelihood, a powerful statistical technique.

Modelling Steps:

* Understanding Problem Statement
* Objective
* Approach
* Data Preparation
  + Missing Treatment
  + Capping/Flooring
  + Exclusion
  + Dummies
  + Interactions
* Train / Test
* Model Selection
* Model Evaluation
* Model Performance Tracking

Lets try and predict if an individual will earn more than $50K using logistic regression based on demographic variables available in the adult data



inputData <- read.csv("http://rstatistics.net/wp-content/uploads/2015/09/adult.csv")

head(inputData)

# Create Training Data

input\_ones <- inputData[which(inputData$ABOVE50K == 1), ] # all 1's

input\_zeros <- inputData[which(inputData$ABOVE50K == 0), ] # all 0's

set.seed(100) # for repeatability of samples

input\_ones\_training\_rows <- sample(1:nrow(input\_ones), 0.7\*nrow(input\_ones)) # 1's for training

input\_zeros\_training\_rows <- sample(1:nrow(input\_zeros), 0.7\*nrow(input\_ones)) # 0's for training. Pick as many 0's as 1's

training\_ones <- input\_ones[input\_ones\_training\_rows, ]

training\_zeros <- input\_zeros[input\_zeros\_training\_rows, ]

trainingData <- rbind(training\_ones, training\_zeros) # row bind the 1's and 0's

# Create Test Data

test\_ones <- input\_ones[-input\_ones\_training\_rows, ]

test\_zeros <- input\_zeros[-input\_zeros\_training\_rows, ]

testData <- rbind(test\_ones, test\_zeros) # row bind the 1's and 0's

logitMod <- glm(ABOVE50K ~ RELATIONSHIP + AGE + CAPITALGAIN + OCCUPATION + EDUCATIONNUM, data=trainingData, family=binomial(link="logit"))

summary(logitMod)

trainingData$p <- predict(logitMod, trainingData, type="response")

testData$p <- predict(logitMod, testData, type="response")

install.packages("car")

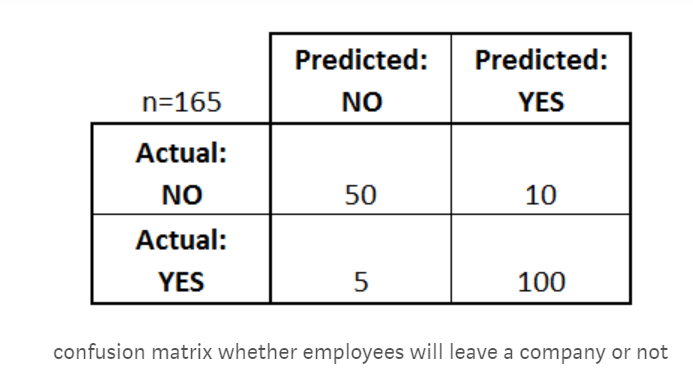
library(car)

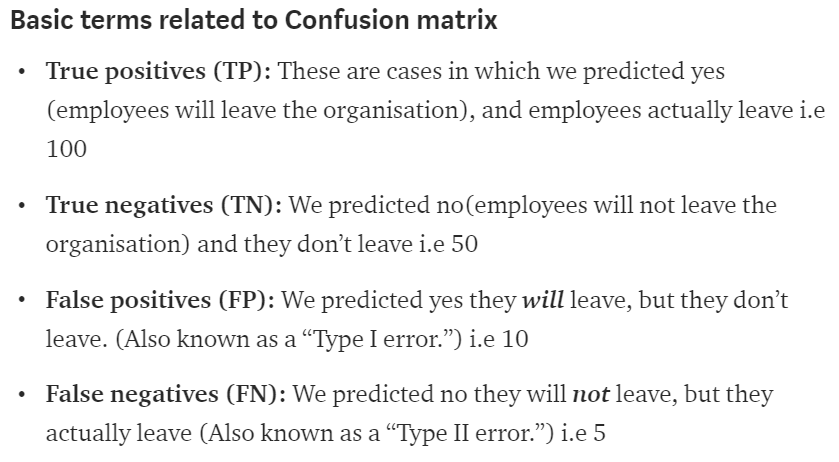
vif(logitMod)

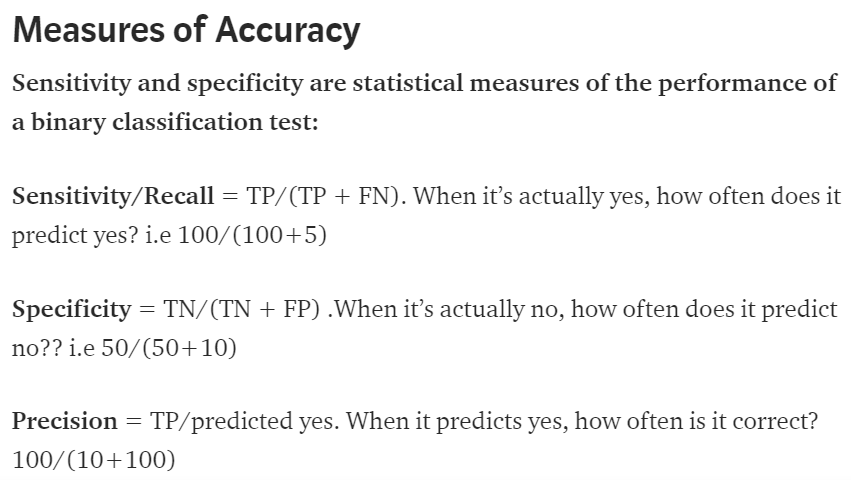
install.packages("InformationValue")

plotROC(trainingData$ABOVE50K, train\_pred)

Concordance(trainingData$ABOVE50K, train\_pred)







sensitivity(trainingData$ABOVE50K, train\_pred)

specificity(trainingData$ABOVE50K, train\_pred)

confusionMatrix(trainingData$ABOVE50K, train\_pred)

somersD(actuals= trainingData$ABOVE50K, predictedScores= trainingData$p)