Machine learning and R Programming

@SCTPL

By Lokesh Singh

Machine learning comes in many different flavors, depending on the algorithm and its objectives. You can divide machine learning algorithms into three main groups based on their purpose:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

SUPERVISED LEARNING

Supervised learning occurs when an algorithm learns from example data and associated target responses that can consist of numeric values or string labels, such as classes or tags, in order to later predict the correct response when posed with new examples. The supervised approach is indeed similar to human learning under the supervision of a teacher. The teacher provides good examples for the student to memorize, and the student then derives general rules from these specific examples.

You need to distinguish between regression problems, whose target is a numeric value, and classification problems, whose target is a qualitative variable, such as a class or a tag. A regression task determines the average prices of houses in the Boston area, and a classification tasks distinguishes between kinds of iris flowers based on their sepal and petal measures.

UNSUPERVISED LEARNING

Unsupervised learning occurs when an algorithm learns from plain examples without any associated response, leaving to the algorithm to determine the data patterns on its own. This type of algorithm tends to restructure the data into something else, such as new features that may represent a class or a new series of uncorrelated values. They are quite useful in providing humans with insights into the meaning of data and new useful inputs to supervised machine learning algorithms.

As a kind of learning, it resembles the methods humans use to figure out that certain objects or events are from the same class, such as by observing the degree of similarity between objects. Some recommendation systems that you find on the web in the form of marketing automation are based on this type of learning.

The marketing automation algorithm derives its suggestions from what you've bought in the past. The recommendations are based on an estimation of what group of customers you resemble the most and then inferring your likely preferences based on that group.

REINFORCEMENT LEARNING

Reinforcement learning occurs when you present the algorithm with examples that lack labels, as in unsupervised learning. However, you can accompany an example with positive or negative feedback according to the solution the algorithm proposes. Reinforcement learning is connected to applications for which the algorithm must make decisions (so the product is prescriptive, not just descriptive, as in unsupervised learning), and the decisions bear consequences. In the human world, it is just like learning by trial and error.

Errors help you learn because they have a penalty added (cost, loss of time, regret, pain, and so on), teaching you that a certain course of action is less likely to succeed than others. An interesting example of reinforcement learning occurs when computers learn to play video games by themselves.

In this case, an application presents the algorithm with examples of specific situations, such as having the gamer stuck in a maze while avoiding an enemy. The application lets the algorithm know the outcome of actions it takes, and learning occurs while trying to avoid what it discovers to be dangerous and to pursue survival. You can have a look at how the company Google DeepMind has created a **reinforcement learning program that plays old Atari's videogames**. When watching the video, notice how the program is initially clumsy and unskilled but steadily improves with training until it becomes a champion.

Machine Learning Algos

R - Linear Regression

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

In Linear Regression these two variables are related through an equation, where exponent (power) of both these variables is 1. Mathematically a linear relationship represents a straight line when plotted as a graph. A non-linear relationship where the exponent of any variable is not equal to 1 creates a curve.

The general mathematical equation for a linear regression is –

```
y = ax + b
```

Following is the description of the parameters used –

- **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

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 –

- Carry out the experiment of 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 the sample data representing the observations –

```
# 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 **lm()** function in linear regression is –

```
lm(formula,data)
```

Following is the description of the parameters used –

- **formula** is a symbol presenting the relation between x and y.
- **data** is the vector on which the formula will be applied.

Create Relationship Model & get the 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 the above code, it produces the following result –

```
Call:
lm(formula = y ~ x)

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

Get the Summary of the Relationship

```
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(summary(relation))
```

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

```
Call:
lm(formula = y ~ x)

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

Syntax

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

```
predict(object, newdata)
```

Following is the description of the parameters used –

- **object** is the formula which is already created using the lm() function.

- **newdata** is the vector containing the new value for predictor variable.

Predict the weight of new persons

```
# The predictor vector.
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

# The resposne 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 we execute the above code, it produces the following result –

```
       1
76.22869
```

Visualize the Regression Graphically

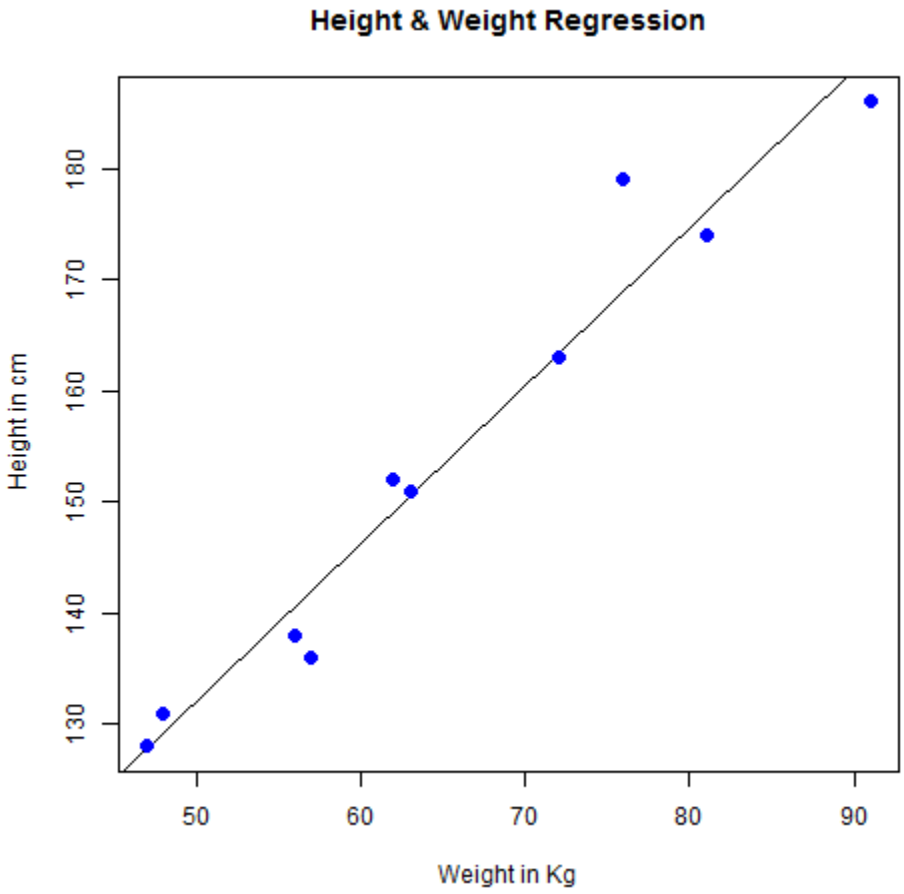
```
# Create the predictor and response variable.
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
relation <- lm(y~x)

# 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 we execute the above code, it produces the following result –



R - Multiple Regression

Multiple regression is an extension of linear regression into relationship between more than two variables. In simple linear relation we have one predictor and one response variable, but in multiple regression we have more than one predictor variable and one response variable.

The general mathematical equation for multiple regression is −

```
y = a + b1x1 + b2x2 +...bnxn
```

Following is the description of the parameters used −

- **y** is the response variable.
- **a, b1, b2...bn** are the coefficients.
- **x1, x2, ...xn** are the predictor variables.

We create the regression model using the **lm()** function in R. The model determines the value of the coefficients using the input data. Next we can predict the value of the response variable for a given set of predictor variables using these coefficients.

lm() Function

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

Syntax

The basic syntax for **lm()** function in multiple regression is −

```
lm(y ~ x1+x2+x3...,data)
```

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between the response variable and predictor variables.

- **data** is the vector on which the formula will be applied.

Example

Input Data

Consider the data set "mtcars" available in the R environment. It gives a comparison between different car models in terms of mileage per gallon (mpg), cylinder displacement("disp"), horse power("hp"), weight of the car("wt") and some more parameters.

The goal of the model is to establish the relationship between "mpg" as a response variable with "disp","hp" and "wt" as predictor variables. We create a subset of these variables from the mtcars data set for this purpose.

```
input <- mtcars[,c("mpg","disp","hp","wt")]

print(head(input))
```

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

```
                  mpg   disp   hp    wt
Mazda RX4         21.0  160    110   2.620
Mazda RX4 Wag     21.0  160    110   2.875
Datsun 710        22.8  108     93   2.320
Hornet 4 Drive    21.4  258    110   3.215
Hornet Sportabout 18.7  360    175   3.440
Valiant           18.1  225    105   3.460
```

Create Relationship Model & get the Coefficients

```
input <- mtcars[,c("mpg","disp","hp","wt")]


# Create the relationship model.

model <- lm(mpg~disp+hp+wt, data = input)


# Show the model.

print(model)


# Get the Intercept and coefficients as vector elements.

cat("# # # # The Coefficient Values # # # ","\n")


a <- coef(model)[1]
```

```
print(a)


Xdisp <- coef(model)[2]
Xhp <- coef(model)[3]
Xwt <- coef(model)[4]


print(Xdisp)
print(Xhp)
print(Xwt)
```

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

```
Call:
lm(formula = mpg ~ disp + hp + wt, data = input)

Coefficients:
(Intercept)         disp           hp           wt
  37.105505      -0.000937        -0.031157    -3.800891

# # # # The Coefficient Values # # #
(Intercept)
   37.10551
         disp
-0.0009370091
         hp
-0.03115655
       wt
-3.800891
```

Create Equation for Regression Model

Based on the above intercept and coefficient values, we create the mathematical equation.

```
Y = a+Xdisp.x1+Xhp.x2+Xwt.x3
or
Y = 37.15+(-0.000937)*x1+(-0.0311)*x2+(-3.8008)*x3
```

Apply Equation for predicting New Values

We can use the regression equation created above to predict the mileage when a new set of values for displacement, horse power and weight is provided.

For a car with disp = 221, hp = 102 and wt = 2.91 the predicted mileage is –

```
Y = 37.15+(-0.000937)*221+(-0.0311)*102+(-3.8008)*2.91 = 22.7104
```

R - 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+...))
```

Following is the description of the parameters used –

- **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)
```

Following is the description of the parameters used –

- **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.

Example

The in-built data set "mtcars" 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.

```
# 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
```

Create Regression Model

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

```
input <- mtcars[,c("am","cyl","hp","wt")]


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
```

Conclusion

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.

kNN Algorithm

This algorithm is a supervised learning algorithm, where the destination is known, but the path to the destination is not. Understanding nearest neighbors forms the quintessence of machine learning. Just like Regression, this algorithm is also easy to learn and apply.

What is kNN Algorithm?

Let's assume we have several groups of labeled samples. The items present in the groups are homogeneous in nature. Now, suppose we have an unlabeled example which needs to be classified into one of the several labeled groups. How do you do that? Unhesitatingly, using kNN Algorithm.

k nearest neighbors is a simple algorithm that stores all available cases and classifies new cases by a majority vote of its k neighbors. This algorithms segregates unlabeled data points into well defined groups.

How to select appropriate k value?

Choosing the number of nearest neighbors i.e. determining the value of k plays a significant role in determining the efficacy of the model. Thus, selection of k will determine how well the data can be utilized to generalize the results of the kNN algorithm. A large k value has benefits which include reducing the variance due to the noisy data; the side effect being developing a bias due to which the learner tends to ignore the smaller patterns which may have useful insights.

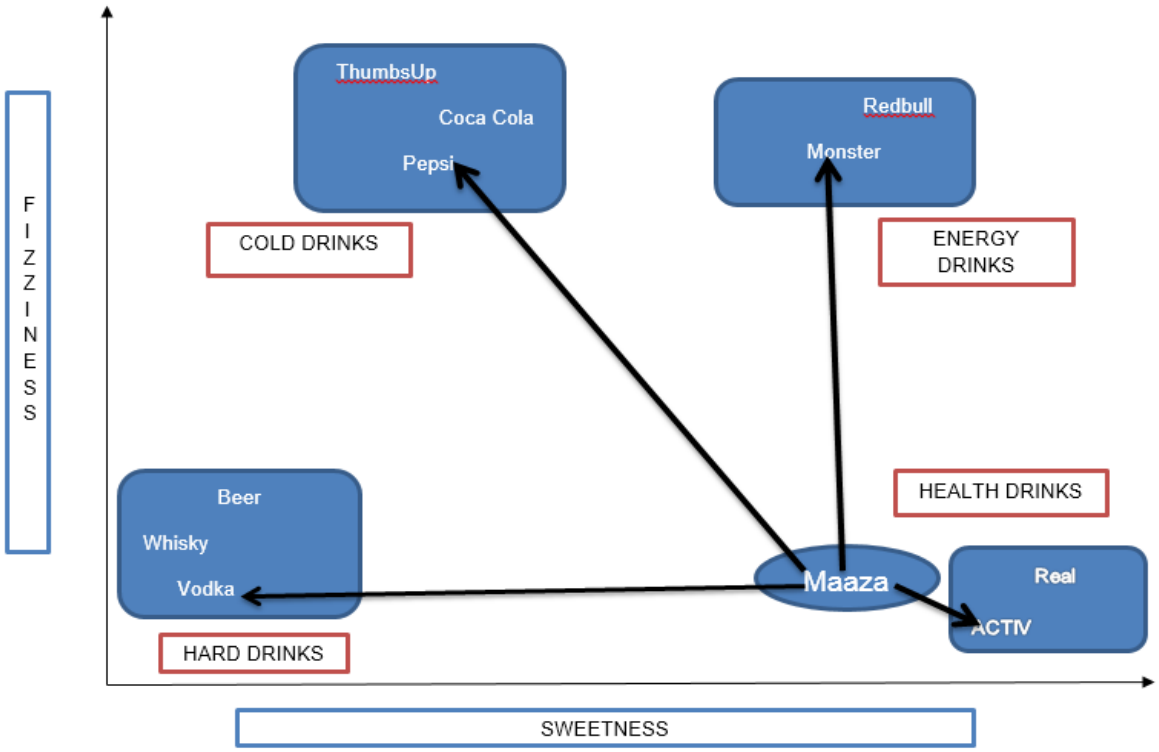
The following example will give you practical insight on selecting the appropriate k value.

Example of kNN Algorithm

Let's consider 10 'drinking items' which are rated on two parameters on a scale of 1 to 10. The two parameters are "sweetness" and "fizziness". This is more of a perception based rating and so may vary between individuals. I would be considering my ratings (which might differ) to take this illustration ahead. The ratings of few items look somewhat as:

Ingredient	Sweetness	Fizziness	Type of Drink
Monster	8	8	Energy booster
ACTIV	9	1	Health drink
Pepsi	4	8	Cold drink
Vodka	2	1	Hard drink

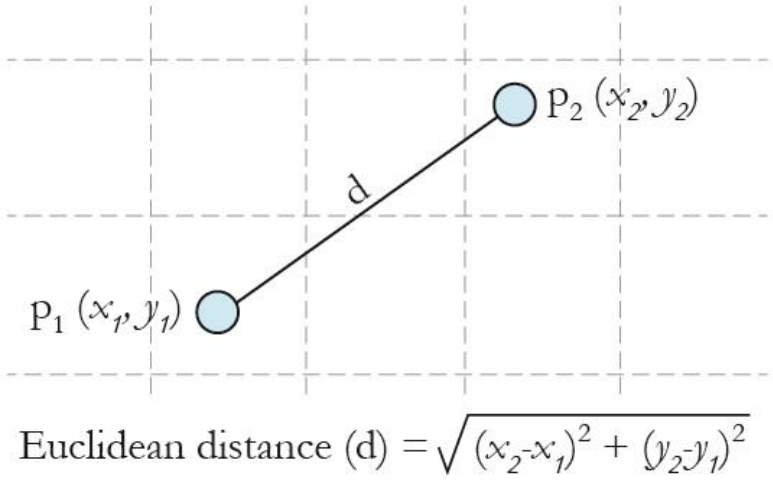
"Sweetness" determines the perception of the sugar content in the items. "Fizziness" ascertains the presence of bubbles in the drink due to the carbon dioxide content in the drink. Again, all these ratings used are based on personal perception and are strictly relative.



From the above figure, it is clear we have bucketed the 10 items into 4 groups namely, 'COLD DRINKS', 'ENERGY DRINKS', 'HEALTH DRINKS' and 'HARD DRINKS'. The question here is, to which group would 'Maaza' fall into? This will be determined by calculating distance.

Calculating Distance

Now, calculating distance between 'Maaza' and its nearest neighbors ('ACTIV', 'Vodka', 'Pepsi' and 'Monster') requires the usage of a distance formula, the most popular being Euclidean distanceformula i.e. the shortest distance between the 2 points which may be obtained using a ruler.



Source: Gamesetmap

Using the co-ordinates of Maaza (8,2) and Vodka (2,1), the distance between 'Maaza' and 'Vodka' can be calculated as:

dist(Maaza,Vodka) = 6.08

Ingredient	Sweetness	Fizziness	Type of Drink	Distance to Maaza
Monster	7	8	Energy booster	6.08
ACTIV	9	1	Health drink	1.41
Pepsi	4	8	Cold drink	7.21
Vodka	2	1	Hard drink	6.08

Using Euclidean distance, we can calculate the distance of Maaza from each of its nearest neighbors. Distance between Maaza and ACTIV being the least, it may be inferred that Maaza is same as ACTIV in nature which in turn belongs to the group of drinks (Health Drinks).

If k=1, the algorithm considers the nearest neighbor to Maaza i.e, ACTIV; if k=3, the algorithm considers '3' nearest neighbors to Maaza to compare the distances (ACTIV, Vodka, Monster) – ACTIV stands the nearest to Maaza.

kNN Algorithm – Pros and Cons

Pros: The algorithm is highly unbiased in nature and makes no prior assumption of the underlying data. Being simple and effective in nature, it is easy to implement and has gained good popularity.

Cons: Indeed it is simple but kNN algorithm has drawn a lot of flake for being extremely simple! If we take a deeper look, this doesn't create a model since there's no abstraction process involved. Yes, the training process is really fast as the data is stored verbatim (hence lazy learner) but the prediction time is pretty high with useful insights missing at times. Therefore, building this algorithm requires time to be invested in data preparation (especially treating the missing data and categorical features) to obtain a robust model.

Case Study: Detecting Prostate Cancer

Machine learning finds extensive usage in pharmaceutical industry especially in detection of oncogenic (cancer cells) growth. R finds application in machine learning to build models to predict the abnormal growth of cells thereby helping in detection of cancer and benefiting the health system.

Let's see the process of building this model using kNN algorithm in R Programming. Below you'll observe I've explained every line of code written to accomplish this task.

Step 1- Data collection

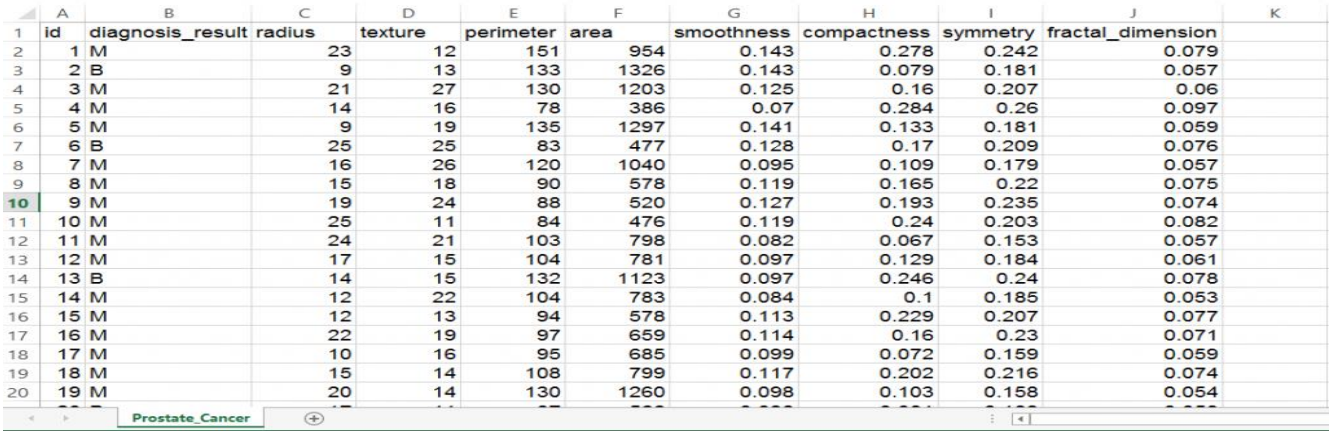
We will use a data set of 100 patients (created solely for the purpose of practice) to implement the knn algorithm and thereby interpreting results .The data set has been prepared keeping in mind the results which are generally obtained from DRE (Digital Rectal Exam). You can download the data set and practice these steps as I explain.

The data set consists of 100 observations and 10 variables (out of which 8 numeric variables and one categorical variable and is ID) which are as follows:

1. Radius
2. Texture
3. Perimeter
4. Area
5. Smoothness
6. Compactness
7. Symmetry
8. Fractal dimension

In real life, there are dozens of important parameters needed to measure the probability of cancerous growth but for simplicity purposes let's deal with 8 of them!

Here's how the data set looks like:



	A	B	C	D	E	F	G	H	I	J	K
1	id	diagnosis_result	radius	texture	perimeter	area	smoothness	compactness	symmetry	fractal_dimension	
2	1	M	23	12	151	954	0.143	0.278	0.242	0.079	
3	2	B	9	13	133	1326	0.143	0.079	0.181	0.057	
4	3	M	21	27	130	1203	0.125	0.16	0.207	0.06	
5	4	M	14	16	78	386	0.07	0.284	0.26	0.097	
6	5	M	9	19	135	1297	0.141	0.133	0.181	0.059	
7	6	B	25	25	83	477	0.128	0.17	0.209	0.076	
8	7	M	16	26	120	1040	0.095	0.109	0.179	0.057	
9	8	M	15	18	90	578	0.119	0.165	0.22	0.075	
10	9	M	19	24	88	520	0.127	0.193	0.235	0.074	
11	10	M	25	11	84	476	0.119	0.24	0.203	0.082	
12	11	M	24	21	103	798	0.082	0.067	0.153	0.057	
13	12	M	17	15	104	781	0.097	0.129	0.184	0.061	
14	13	B	14	15	132	1123	0.097	0.246	0.24	0.078	
15	14	M	12	22	104	783	0.084	0.1	0.185	0.053	
16	15	M	12	13	94	578	0.113	0.229	0.207	0.077	
17	16	M	22	19	97	659	0.114	0.16	0.23	0.071	
18	17	M	10	16	95	685	0.099	0.072	0.159	0.059	
19	18	M	15	14	108	799	0.117	0.202	0.216	0.074	
20	19	M	20	14	130	1260	0.098	0.103	0.158	0.054	

Prostate_Cancer

Step 2- Preparing and exploring the data

```
[Previously saved workspace restored]

> setwd("C:/Users/Payal/Desktop/KNN")
> prc <- read.csv("Prostrate_Cancer.csv",stringsAsFactors = FALSE)
> str(prc)
'data.frame':   100 obs. of  10 variables:
 $ id                : int  1 2 3 4 5 6 7 8 9 10 ...
 $ diagnosis_result : chr  "M" "B" "M" "M" ...
 $ radius            : int  23 9 21 14 9 25 16 15 19 25 ...
 $ texture           : int  12 13 27 16 19 25 26 18 24 11 ...
 $ perimeter         : int  151 133 130 78 135 83 120 90 88 84 ...
 $ area              : int  954 1326 1203 386 1297 477 1040 578 520 476 ...
 $ smoothness        : num  0.143 0.143 0.125 0.07 0.141 0.128 0.095 0.119 0.127 0.119 ...
 $ compactness       : num  0.278 0.079 0.16 0.284 0.133 0.17 0.109 0.165 0.193 0.24 ...
 $ symmetry          : num  0.242 0.181 0.207 0.26 0.181 0.209 0.179 0.22 0.235 0.203 ...
 $ fractal dimension: num  0.079 0.057 0.06 0.097 0.059 0.076 0.057 0.075 0.074 0.082 ...
>
```

Let's make sure that we understand every line of code before proceeding to the next stage:

```
setwd("C:/Users/Payal/Desktop/KNN")    #Using this command, we've imported the 'Prostate_Cancer.csv' data file. This command is used to point to the folder containing the required file. Do keep in mind, that it's a common mistake to use "\" instead of "/" after the setwd command.

prc <- read.csv("Prostate_Cancer.csv",stringsAsFactors = FALSE)    #This command imports the required data set and saves it to the prc data frame.

stringsAsFactors = FALSE   #This command helps to convert every character vector to a factor wherever it makes sense.

str(prc)    #We use this command to see whether the data is structured or not.
```

We find that the data is structured with 10 variables and 100 observations. If we observe the data set, the first variable 'id' is unique in nature and can be removed as it does not provide useful information.

```
> prc <- prc[-1]
> head(prc)
  diagnosis_result radius texture perimeter area smoothness compactness symmetry fractal_
1                M     23      12       151  954      0.143       0.278    0.242
2                B      9      13       133 1326      0.143       0.079    0.181
3                M     21      27       130 1203      0.125       0.160    0.207
4                M     14      16        78  386      0.070       0.284    0.260
5                M      9      19       135 1297      0.141       0.133    0.181
6                B     25      25        83  477      0.128       0.170    0.209
>
```

```
prc <- prc[-1]  #removes the first variable(id) from the data set.
```

The data set contains patients who have been diagnosed with either Malignant (M) or Benign (B) cancer

```
table(prc$diagnosis_result)  # it helps us to get the numbers of patients
```

(The variable diagnosis_result is our target variable i.e. this variable will determine the results of the diagnosis based on the 8 numeric variables)

In case we wish to rename B as"Benign" and M as "Malignant" and see the results in the percentage form, we may write as:

```
prc$diagnosis <- factor(prc$diagnosis_result, levels = c("B", "M"), labels = c("Benig
n", "Malignant"))


round(prop.table(table(prc$diagnosis)) * 100, digits = 1)  # it gives the result in t
he percentage form rounded of to 1 decimal place( and so it's digits = 1)
```

```
> prc$diagnosis <- factor(prc$diagnosis_result, levels = c("B", "M"), labels = c("Benign",
> round(prop.table(table(prc$diagnosis)) * 100, digits = 1)

   Benign Malignant
       38        62
>
```

Normalizing numeric data

This feature is of paramount importance since the scale used for the values for each variable might be different. The best practice is to normalize the data and transform all the values to a common scale.

```
normalize <- function(x) {

return ((x - min(x)) / (max(x) - min(x))) }
```

Once we run this code, we are required to normalize the numeric features in the data set. Instead of normalizing each of the 8 individual variables we use:

```
prc_n <- as.data.frame(lapply(prc[2:9], normalize))
```

The first variable in our data set (after removal of id) is 'diagnosis_result' which is not numeric in nature. So, we start from 2nd variable. The function lapply() applies normalize() to each feature in the data frame. The final result is stored to prc_n data frame using as.data.frame() function

Let's check using the variable 'radius' whether the data has been normalized.

```
summary(prc_n$radius)
```

```
> normalize <- function(x) {
+  return ((x - min(x)) / (max(x) - min(x)))
+  }
> prc_n <- as.data.frame(lapply(prc[2:9], normalize))
> summary(prc_n$radius)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.1875  0.5000  0.4906  0.7500  1.0000
> |
```

The results show that the data has been normalized. Do try with the other variables such as perimeter, area etc.

Creating training and test data set

The kNN algorithm is applied to the training data set and the results are verified on the test data set.

For this, we would divide the data set into 2 portions in the ratio of 65: 35 (assumed) for the training and test data set respectively. You may use a different ratio altogether depending on the business requirement!

We shall divide the prc_n data frame into prc_train and prc_test data frames

```
prc_train <- prc_n[1:65,]


prc_test <- prc_n[66:100,]
```

A blank value in each of the above statements indicate that all rows and columns should be included.

Our target variable is ‘diagnosis_result’ which we have not included in our training and test data sets.

```
prc_train_labels <- prc[1:65, 1]
```

```
prc_test_labels <- prc[66:100, 1]   #This code takes the diagnosis factor in column 1
of the prc data frame and on turn creates prc_train_labels and prc_test_labels data f
rame.
```

Step 3 – Training a model on data

The knn () function needs to be used to train a model for which we need to install a package 'class'. The knn() function identifies the k-nearest neighbors using Euclidean distance where k is a user-specified number.

You need to type in the following commands to use knn()

```
install.packages("class")

library(class)
```

Now we are ready to use the knn() function to classify test data

```
prc_test_pred <- knn(train = prc_train, test = prc_test,cl = prc_train_labels, k=10)
```

The value for k is generally chosen as the square root of the number of observations.

knn() returns a factor value of predicted labels for each of the examples in the test data set which is then assigned to the data frame prc_test_pred

```
>  prc_train <- prc_n[1:65, ]
>  prc_test <- prc_n[66:100, ]
> prc_test_labels <- prc[66:100, 1]
> prc_train_labels <- prc[1:65, 1]
> prc_test_labels <- prc[66:100, 1]
> prc_test_pred <- knn(train = prc_train, test = prc_test,cl = prc_train_labels, k=10)
Error: could not find function "knn"
> install.packages("class")
Installing package into 'C:/Users/Payal/Documents/R/win-library/3.1'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
trying URL 'http://mirrors.xmu.edu.cn/CRAN/bin/windows/contrib/3.1/class_7.3-13.zip'
Content type 'application/zip' length 100211 bytes (97 Kb)
opened URL
downloaded 97 Kb

package 'class' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Payal\AppData\Local\Temp\RtmpkPNcE8\downloaded_packages
> library(class)
Warning message:
package 'class' was built under R version 3.1.3
> prc_test_pred <- knn(train = prc_train, test = prc_test,cl = prc_train_labels, k=10)
>
```

Step 4 – Evaluate the model performance

We have built the model but we also need to check the accuracy of the predicted values in prc_test_pred as to whether they match up with the known values in prc_test_labels. To ensure this, we need to use the CrossTable() function available in the package 'gmodels'.

We can install it using:

```
install.packages("gmodels")
```

```
> library(gmodels)
Warning message:
package 'gmodels' was built under R version 3.1.3
> CrossTable(x = prc_test_labels, y = prc_test_pred,prop.chisq=FALSE)


   Cell Contents
|-------------------------|
|                       N |
|           N / Row Total |
|           N / Col Total |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  35
```

prc_test_labels	prc_test_pred: B	M	Row Total
B	5	14	19
	0.263	0.737	0.543
	1.000	0.467	
	0.143	0.400	
M	0	16	16
	0.000	1.000	0.457
	0.000	0.533	
	0.000	0.457	
Column Total	5	30	35
	0.143	0.857	

The test data consisted of 35 observations. Out of which 5 cases have been accurately predicted (TN->True Negatives) as Benign (B) in nature which constitutes 14.3%. Also, 16 out of 35 observations were accurately predicted (TP-> True Positives) as Malignant (M) in nature which constitutes 45.7%. Thus a total of 16 out of 35 predictions where TP i.e, True Positive in nature.

There were no cases of False Negatives (FN) meaning no cases were recorded which actually are malignant in nature but got predicted as benign. The FN's if any poses a potential threat for the same reason and the main focus to increase the accuracy of the model is to reduce FN's.

There were 14 cases of False Positives (FP) meaning 14 cases were actually benign in nature but got predicted as malignant.

The total accuracy of the model is 60 %((TN+TP)/35) which shows that there may be chances to improve the model performance

Step 5 – Improve the performance of the model

This can be taken into account by repeating the steps 3 and 4 and by changing the k-value. Generally, it is the square root of the observations and in this case we took k=10 which is a perfect square root of 100.The k-value may be fluctuated in and around the value of 10 to check the increased accuracy of the model. Do try it out with values of your choice to increase the accuracy! Also remember, to keep the value of FN's as low as possible.

K MEAN CLUSERING

K-Means is a clustering approach that belogs to the class of unsupervised statistical learning methods. K-Means is very popular in a variety of domains. In biology it is often used to find structure in DNA-related data or subgroups of similar tissue samples to identify cancer cohorts. In marketing, K-Means is often used to create market/customer/product segments.

The general idea of a clustering algorithm is to partition a given dataset into distinct, exclusive clusters so that the data points in each group are quite similar to each other.

One of the first steps in building a K-Means clustering work is to define the number of clusters to work with. Subsequently, the algorithm assigns each individual data point to one of the clusters in a random fashion. The underlying idea of the algorithm is that a good cluster is the one which contains the smallest possible *within-cluster* variation of all observations in relation to each other. The most common way to define this variation is using the **squared Euclidean distance**. This process of identifying groups of similar data points can be a relatively complex task since there is a very large number of ways to partion data points into clusters.

Generally, the way K-Means algorithms work is via an iterative refinement process:

1. Each data point is randomly assigned to a cluster (number of clusters is given before hand).
2. Each cluster's centroid (mean within cluster) is calculated.
3. Each data point is assigned to its nearest centroid (iteratively to minimise the within-cluster variation) until no major differences are found.

Let's have a look at an example in R using the **Chatterjee-Price Attitude Data** from the `library(datasets)` package. The dataset is a survey of clerical employees of a large financial organization. The data are aggregated from questionnaires of approximately 35 employees for each of 30 (randomly selected) departments. The numbers give the percent proportion of favourable responses to seven questions in each department. For more details, see `?attitude`.

```
# Load necessary libraries
library(datasets)


# Inspect data structure
str(attitude)
## 'data.frame':    30 obs. of  7 variables:
##  $ rating    : num  43 63 71 61 81 43 58 71 72 67 ...
##  $ complaints: num  51 64 70 63 78 55 67 75 82 61 ...
##  $ privileges: num  30 51 68 45 56 49 42 50 72 45 ...
##  $ learning  : num  39 54 69 47 66 44 56 55 67 47 ...
##  $ raises    : num  61 63 76 54 71 54 66 70 71 62 ...
##  $ critical  : num  92 73 86 84 83 49 68 66 83 80 ...
##  $ advance   : num  45 47 48 35 47 34 35 41 31 41 ...
# Summarise data
summary(attitude)
```

```
##      rating        complaints     privileges       learning
##  Min.   :40.00   Min.   :37.0   Min.   :30.00   Min.   :34.00
##  1st Qu.:58.75   1st Qu.:58.5   1st Qu.:45.00   1st Qu.:47.00
##  Median :65.50   Median :65.0   Median :51.50   Median :56.50
##  Mean   :64.63   Mean   :66.6   Mean   :53.13   Mean   :56.37
##  3rd Qu.:71.75   3rd Qu.:77.0   3rd Qu.:62.50   3rd Qu.:66.75
##  Max.   :85.00   Max.   :90.0   Max.   :83.00   Max.   :75.00
##      raises          critical        advance
##  Min.   :43.00   Min.   :49.00   Min.   :25.00
##  1st Qu.:58.25   1st Qu.:69.25   1st Qu.:35.00
##  Median :63.50   Median :77.50   Median :41.00
##  Mean   :64.63   Mean   :74.77   Mean   :42.93
##  3rd Qu.:71.00   3rd Qu.:80.00   3rd Qu.:47.75
##  Max.   :88.00   Max.   :92.00   Max.   :72.00
```

As we've seen, this data gives the percent of favourable responses for each department. For example, in the summary output above we can see that for the variable **privileges** among all 30 departments the minimum percent of favourable responses was 30 and the maximum was 83. In other words, one department had only 30% of responses favourable when it came to assessing 'privileges' and one department had 83% of favourable responses when it came to assessing 'privileges', and a lot of other favourable response levels in between.

When performing clustering, some important concepts must be tackled. One of them is how to deal with data that contains multiple (or more than 2) variables. In such cases, one option would be to perform Principal Component Analysis (PCA) and then plot the first two vectors and maybe additionally apply K-Means. Other checks to be made are whether the data in hand should be standardized, whether the number of clusters obtained are truly representing the underlying pattern found in the data, whether there could be other clustering algorithms or parameters to be taken, etc. It is often recommended to perform clustering algorithms with different approaches and preferably test the clustering results with independent datasets. Particularly, it is very important to be careful with the way the results are reported and used.

For simplicity, we're not going to tackle most of these concerns in this example but they should always be part of a more robust work.

In light of the example, we'll take a subset of the attitude dataset and consider only two variables in our K-Means clustering exercise. So imagine that we would like to cluster the attitude dataset with the responses from all 30 departments when it comes to 'privileges' and 'learning' and we would like to understand whether there are commonalities among certain departments when it comes to these two variables.

```
# Subset the attitude data
dat = attitude[,c(3,4)]
```

```
# Plot subset data
plot(dat, main = "% of favourable responses to
     Learning and Privilege", pch =20, cex =2)
```

With the data subset and the plot above we can see how each department's score behave across Privilege and Learning compare to each other. In the most simplistic sense, we can apply K-Means clustering to this data set and try to assign each department to a specific number of clusters that are "similar".

Let's use the **kmeans** function from R base stats package:

```
# Perform K-Means with 2 clusters
set.seed(7)
km1 = kmeans(dat, 2, nstart=100)


# Plot results
plot(dat, col =(km1$cluster +1) , main="K-Means result with 2 clusters", pch=
20, cex=2)
```

As mentioned before, one of the key decisions to be made when performing K-Means clustering is to decide on the numbers of clusters to use. In practice, there is no easy answer and it's important to try different ways and numbers of clusters to decide which options is the most useful, applicable or interpretable solution.

In the plot above, we randomly chose the number of clusters to be 2 for illustration purposes only.

However, one solution often used to identifiy the optimal number of clusters is called the **Elbow** method and it involves observing a set of possible numbers of clusters relative to how they minimise the within-cluster sum of squares. In other words, the Elbow method examines the within-cluster dissimilarity as a function of the number of clusters. Below is a visual representation of the method:

```
# Check for the optimal number of clusters given the data


mydata <- dat
wss <- (nrow(mydata)-1)*sum(apply(mydata,2,var))
  for (i in 2:15) wss[i] <- sum(kmeans(mydata,
                                       centers=i)$withinss)
plot(1:15, wss, type="b", xlab="Number of Clusters",
     ylab="Within groups sum of squares",
     main="Assessing the Optimal Number of Clusters with the Elbow Method",
```

```
     pch=20, cex=2)
```

With the Elbow method, the solution criterion value (within groups sum of squares) will tend to decrease substantially with each successive increase in the number of clusters. Simplistically, an optimal number of clusters is identified once a “kink” in the line plot is observed. As you can grasp, identifying the point in which a “kink” exists is not a very objective approach and is very prone to heuristic processes.

But from the example above, we can say that after 6 clusters the observed difference in the within-cluster dissimilarity is not substantial. Consequently, we can say with some reasonable confidence that the optimal number of clusters to be used is 6.

Assuming this assertion is valid, we can go on and apply the identified number of clusters onto the K-Means algorithm and plot the results:

```
# Perform K-Means with the optimal number of clusters identified from the Elb
ow method

set.seed(7)

km2 = kmeans(dat, 6, nstart=100)


# Examine the result of the clustering algorithm

km2
```

```
## K-means clustering with 6 clusters of sizes 4, 2, 8, 6, 8, 2

##

## Cluster means:

##   privileges learning

## 1   54.50000   71.000

## 2   75.50000   49.500

## 3   47.62500   45.250

## 4   67.66667   69.000

## 5   46.87500   57.375

## 6   31.50000   36.500

##

## Clustering vector:

##  [1] 6 5 4 3 1 3 5 5 4 3 5 3 3 2 1 1 4 4 5 2 6 5 3 5 3 4 1 3 4 5

##

## Within cluster sum of squares by cluster:

## [1]  71.0000 153.0000 255.3750 133.3333 244.7500  17.0000

##  (between_SS / total_SS =  89.5 %)
```

```
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"
## [5] "tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"
```

```
# Plot results
plot(dat, col =(km2$cluster +1) , main="K-Means result with 6 clusters", pch=
20, cex=2)
```

From the results above we can see that there is a relatively well defined set of groups of departments that are relatively distinct when it comes to answering favourably around Privileges and Learning in the survey. It is only natural to think the next steps from this sort of output. One could start to devise strategies to understand why certain departments rate these two different measures the way they do and what to do about it. But we will leave this to another exercise.

This is a walk-through of a customer segmentation process using R's skmeans package to perform k-medians clustering. The dataset examined is that used in chapter 2 of John Foreman's book, Data Smart.
The approach followed is that outlined by the author. The major difference is that the author, as per his teaching objectives, built his solution manually in Excel. On the other hand, we are free to take advantage of R's off-the-shelf packages to solve the problem and we do so here.

This walk-through includes R function calls to calculate silhouette values as measures of segmentation effectiveness.

A note on the silhouette

The silhouette calculation for an assigned data point requires three summary statistics:

- A – the average distance to the data points in its nearest neighboring cluster
- B – the average distance to the data points in its assigned cluster
- C – the maximum of A and B

The silhouette for a data point is (A-B)/C. It is a measure of the robustness of the assignment. A value close to 1 indicates that a data point is very well suited to its cluster. A value close to (or less than) zero indicates ambiguity about the cluster assignment.

The overall silhouette value is the mean of the silhouette values of the individual data points.

R CODE

```
# Load R packages
library(skmeans)  # used for k-medians algorithm
library(cluster)  # required for calling the silhouette function
library(dplyr)  # an excellent tool for for summarizing, ordering and filtering data features
# Read in the sales promotion dataset. Remove meta columns, convert NA values to zeroes
kmcDF <- read.csv(".\\wineKMC_matrix.csv")  #reads in data as a dataframe
wineDF <- t(kmcDF[,-c(1,2,3,4,5,6,7)])  # new variable, metadata columns removed, dataframe transposed
wineDF[is.na(wineDF)] <- 0  # replaces blank entries with zeros
wineMatrix <-as.matrix(wineDF)  #converts the dataframe to type matrix

# Segment the customers into 5 clusters
partition <- skmeans(wineMatrix, 5)

# Look at the segmentation outcome summary
partition  # returns a summary statement for the process
partition$cluster  # returns a vector showing cluster assignment for each customer

# Create a vector of customer names for each cluster
cluster_1 <- names(partition$cluster[partition$cluster == 1])
cluster_2 <- names(partition$cluster[partition$cluster == 2])
cluster_3 <- names(partition$cluster[partition$cluster == 3])
cluster_4 <- names(partition$cluster[partition$cluster == 4])
```

```
cluster_5 <- names(partition$cluster[partition$cluster == 5])

# Examine one of the clusters, as an example
cluster_1
```

The skmeans object, partition, informs us that the result of the segmentation is:
a hard spherical k-means partition of 100 objects into 5 classes.
Class sizes: 14, 18, 16, 29, 23

The customers assigned to cluster_1 are shown here. You may see variation in cluster assignments and sizes from segmentation run to segmentation run.
[1] “Butler” “Clark” “Collins” “Cooper” “Davis” “Fisher”
[7] “Garcia” “Gomez” “Hall” “Howard” “Jackson” “Lopez”
[13] “Martin” “Martinez” “Parker” “Powell” “Reed” “Sanchez”
[19] “Sanders” “Thomas” “Thompson” “Ward” “White”

```
# Examine characteristics of each cluster using the aggregation function to sum the number of purchases for each promotion by cluster
clusterCounts <- t(aggregate(wineDF, by=list(partition$cluster), sum)[,2:33]) # taken directly from Data Smart
clusterCounts <- cbind(kmcDF[,c(1:7)], clusterCounts) # add back the meta data columns

# The arrange function in the dplyr package is used to view the characteristics of the differnt clusters
View(arrange(clusterCounts, -clusterCounts$"1"))
View(arrange(clusterCounts, -clusterCounts$"2"))
View(arrange(clusterCounts, -clusterCounts$"3"))
View(arrange(clusterCounts, -clusterCounts$"4"))
View(arrange(clusterCounts, -clusterCounts$"5"))
```

Some sample results are shown:

Offer..	Campaign	Varietal	Minimum.Qty..kg.	Discount....	Origin	Past.Peak	1	2	3	4	5
11	May	Champagne	72	85	France	FALSE	12	0	1	0	0
9	April	Chardonnay	144	57	Chile	FALSE	8	0	0	2	0
22	August	Champagne	72	63	France	FALSE	8	0	2	11	0
1	January	Malbec	72	56	France	FALSE	7	0	0	2	1
4	February	Champagne	72	48	France	TRUE	6	0	1	5	0
25	October	Cabernet Sauvignon	72	59	Oregon	TRUE	6	0	0	0	0
28	November	Cabernet Sauvignon	12	56	France	TRUE	5	1	0	0	0
30	December	Malbec	6	54	France	FALSE	5	11	5	1	0
2	January	Pinot Noir	72	17	France	FALSE	4	0	0	0	6

Example 1: cluster_1 has a liking for French wines

Offer..	Campaign	Varietal	Minimum.Qty..kg.	Discount....	Origin	Past.Peak	1	2	3	4	5
7	March	Prosecco	6	40	Australia	TRUE	0	14	2	3	0
29	November	Pinot Grigio	6	87	France	FALSE	1	12	4	0	0
30	December	Malbec	6	54	France	FALSE	5	11	5	1	0
18	July	Espumante	6	50	Oregon	FALSE	0	9	4	1	0
13	May	Merlot	6	43	Chile	FALSE	0	4	2	0	0
8	March	Espumante	6	45	South Africa	FALSE	0	3	16	1	0
3	February	Espumante	144	32	Oregon	TRUE	0	1	3	2	0
10	April	Prosecco	72	52	California	FALSE	0	1	1	4	1
12	May	Prosecco	72	83	Australia	FALSE	3	1	0	0	1

Example 2: cluster_2 favors low-volume purchases

Offer..	Campaign	Varietal	Minimum.Qty..kg.	Discount....	Origin	Past.Peak	1	2	3	4	5
24	September	Pinot Noir	6	34	Italy	FALSE	0	0	0	0	12
26	October	Pinot Noir	144	83	Australia	FALSE	2	0	0	1	12
17	July	Pinot Noir	12	47	Germany	FALSE	0	0	0	0	7
2	January	Pinot Noir	72	17	France	FALSE	4	0	0	0	6
1	January	Malbec	72	56	France	FALSE	7	0	0	2	1
10	April	Prosecco	72	52	California	FALSE	0	1	1	4	1

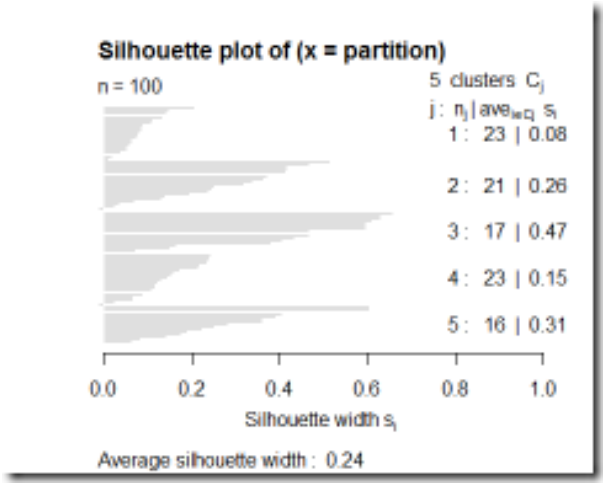
Example 3: cluster_5 members like to drink pinot noir

Computing the silhouette with R

```
# Compare performance of the k=5 clustering process with the k=4 clustering using the silhouette function. The closer the value is to 1, the better.

silhouette_k5 <- silhouette(partition)
summary(silhouette_k5)
plot(silhouette_k5)
```

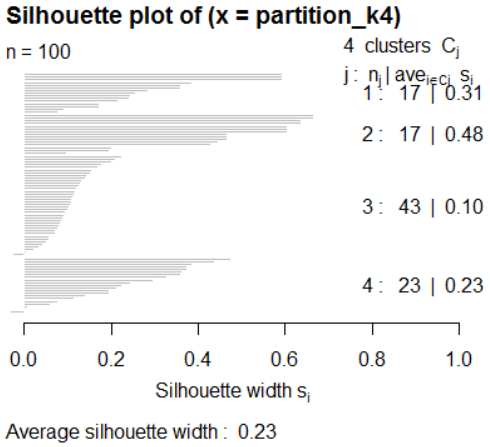
Silhouette of 100 units in 5 clusters from silhouette.skmeans(x = partition) :
Cluster sizes and average silhouette widths:
23 21 17 23 16
0.08447426 0.26400515 0.46935604 0.14643094 0.30914071
Individual silhouette widths:
Min. 1st Qu. Median Mean 3rd Qu. Max.
-0.01138 0.08694 0.17350 0.23780 0.37390 0.65650




```
partition_k4 <- skmeans(wineMatrix, 4)
silhouette_k4 <- silhouette(partition_k4)
plot(silhouette_k4)
summary(silhouette_k4)
```

Silhouette of 100 units in 4 clusters from silhouette.skmeans(x = partition_k4) :
Cluster sizes and average silhouette widths:

17 17 43 23
0.3077986 0.4817904 0.1001556 0.2274353
Individual silhouette widths:
Min. 1st Qu. Median Mean 3rd Qu. Max.
-0.04095 0.09243 0.16950 0.22960 0.36070 0.66550



Conclusion

Typical silhouette values for repeated runs with k=5 and k=4 were 0.23 and 0.24 respectively and we can conclude that the segmentations were about equally effective. In this case, visual examination of the segment metadata is useful for selecting which of the k=4 and k=5 partitions better matches the requirements of the business.

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.

Examples of use of decision tress is – predicting an email as spam or not spam, predicting of a tumor is cancerous or predicting a loan as a good or bad credit risk based on the factors in each of these. 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 arrive at a decision on the category (yes/No, spam/not spam) of the data.

The R package **"party"** is used to create decision trees.

Install R Package

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

```
install.packages("party")
```

The package "party" has the function **ctree()** which is used to create and analyze decison tree.

Syntax

The basic syntax for creating a decision tree in R is –

```
ctree(formula, data)
```

Following is the description of the parameters used –

- **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.

Here is the sample data.

```
# 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, it produces the following result and chart –

```
  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
.............................
.............................
```

Example

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

```
# Load the party package. It will automatically load other dependent packages.
library(party)


# 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 –

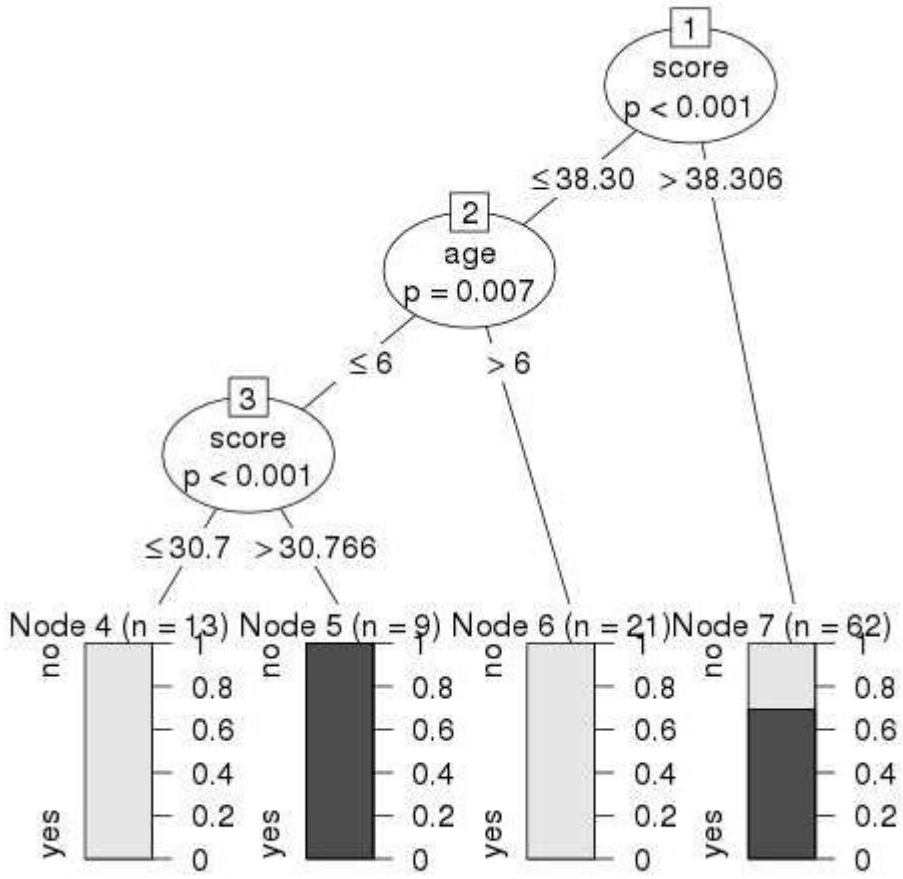
```
null device
          1
Loading required package: methods
Loading required package: grid
Loading required package: mvtnorm
Loading required package: modeltools
Loading required package: stats4
Loading required package: strucchange
Loading required package: zoo

Attaching package: ‘zoo’

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

    as.Date, as.Date.numeric

Loading required package: sandwich
```



Conclusion

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.

R - 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.

Install R Package

Use the below command in R console to install the package. You 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)
```

Following is the description of the parameters used –

- **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.

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 we execute the above code, it produces the following result and chart –

```
  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
.............................
.............................
```

Example

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

```
# Load the party package. It will automatically load other required packages.
library(party)
library(randomForest)


# 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 we execute the above code, it produces the following result –

```
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
```

Conclusion

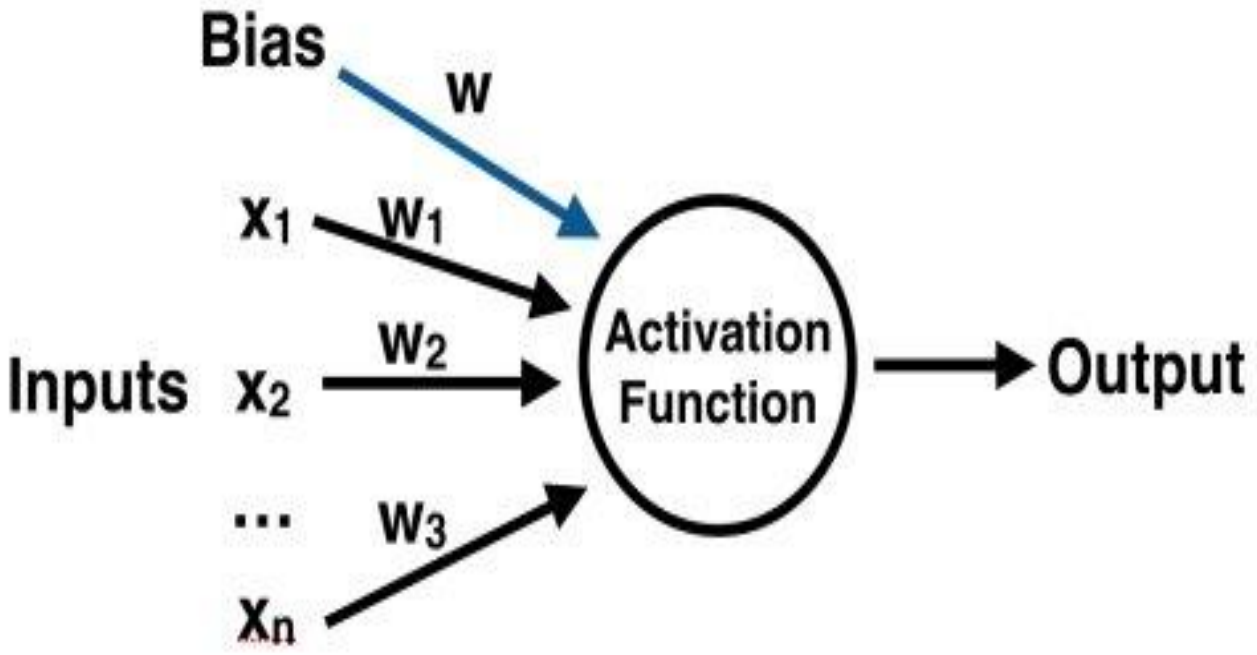
From the random forest shown above we can conclude that the shoesize and score are the important factors deciding if someone is a native speaker or not. Also the model has only 1% error which means we can predict with 99% accuracy.

Neural Networks

Neural Networks are a machine learning framework that attempts to mimic the learning pattern of natural biological neural networks. Biological neural networks have interconnected neurons with dendrites that receive inputs, then based on these inputs they produce an output signal through an axon to another neuron. We will try to mimic this process through the use of Artificial Neural Networks (ANN), which we will just refer to as neural networks from now on. The process of creating a neural network begins with the most basic form, a single perceptron.

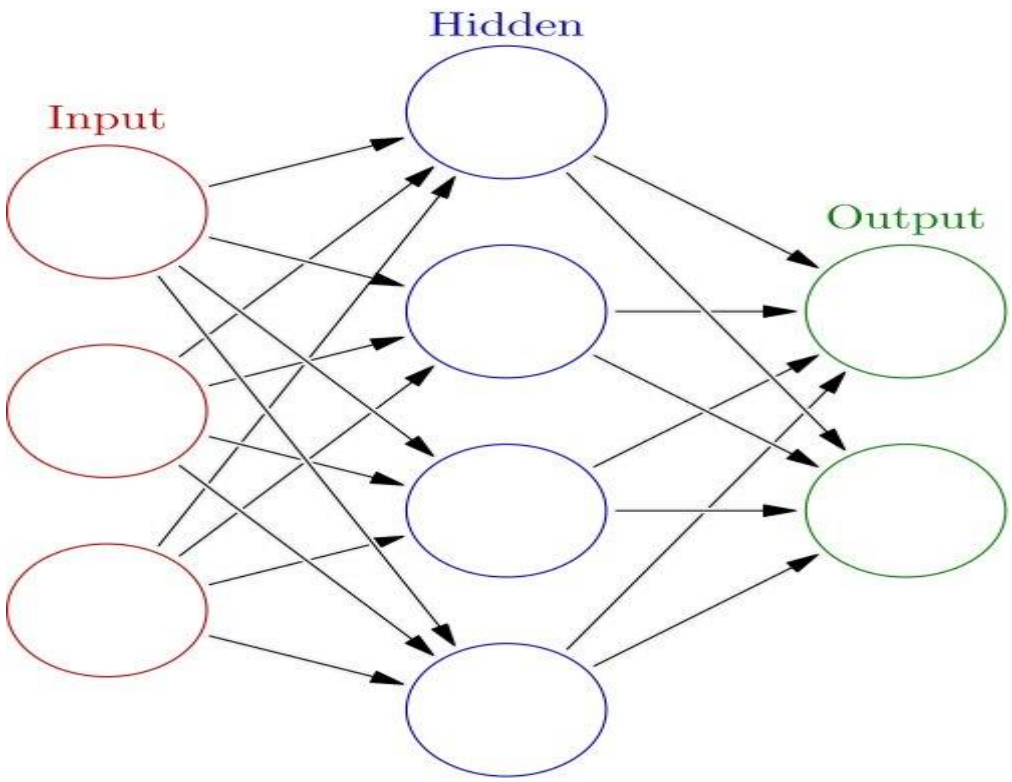
The Perceptron

Let's start our discussion by talking about the Perceptron! A perceptron has one or more inputs, a bias, an activation function, and a single output. The perceptron receives inputs, multiplies them by some weight, and then passes them into an activation function to produce an output. There are many possible activation functions to choose from, such as the logistic function, a trigonometric function, a step function etc. We also make sure to add a bias to the perceptron, this avoids issues where all inputs could be equal to zero (meaning no multiplicative weight would have an effect). Check out the diagram below for a visualization of a perceptron:



Once we have the output we can compare it to a known label and adjust the weights accordingly (the weights usually start off with random initialization values). We keep repeating this process until we have reached a maximum number of allowed iterations, or an acceptable error rate.

To create a neural network, we simply begin to add layers of perceptrons together, creating a multi-layer perceptron model of a neural network. You'll have an input layer which directly takes in your feature inputs and an output layer which will create the resulting outputs. Any layers in between are known as hidden layers because they don't directly "see" the feature inputs or outputs. For a visualization of this check out the diagram below (source: Wikipedia).



Let's move on to actually creating a neural network in R!

Data

We'll use ISLR's built in College Data Set which has several features of a college and a categorical column indicating whether or not the School is Public or Private.

```
#install.packages('ISLR')
library(ISLR)

print(head(College,2))
```

```
        Private Apps Accept Enroll Top10perc Top25perc
```

```
Abilene Christian University     Yes 1660   1232    721        23        52
Adelphi University               Yes 2186   1924    512        16        29
                            F.Undergrad P.Undergrad Outstate Room.Board
Books
Abilene Christian University        2885         537     7440       3300
450
Adelphi University                  2683        1227    12280       6450
750
                            Personal PhD Terminal S.F.Ratio perc.alumni
Expend
Abilene Christian University     2200  70       78      18.1          12
7041
Adelphi University               1500  29       30      12.2          16
10527
                            Grad.Rate
Abilene Christian University        60
Adelphi University                  56
```

Data Preprocessing

It is important to normalize data before training a neural network on it. The neural network may have difficulty converging before the maximum number of iterations allowed if the data is not normalized. There are a lot of different methods for normalization of data. We will use the built-in scale() function in R to easily accomplish this task.
Usually it is better to scale the data from 0 to 1, or -1 to 1. We can specify the center and scale as additional arguments in the scale() function. For example:

```
# Create Vector of Column Max and Min Values
maxs <- apply(College[,2:18], 2, max)
mins <- apply(College[,2:18], 2, min)

# Use scale() and convert the resulting matrix to a data frame
scaled.data <- as.data.frame(scale(College[,2:18],center = mins, scale = maxs
- mins))

# Check out results
print(head(scaled.data,2))
```

```
                                 Apps        Accept        Enroll
Abilene Christian University 0.03288692646 0.04417701272 0.10791253736
Adelphi University           0.04384229271 0.07053088583 0.07503539405
                               Top10perc    Top25perc   F.Undergrad
Abilene Christian University 0.2315789474 0.4725274725 0.08716353479
Adelphi University           0.1578947368 0.2197802198 0.08075165058
                              P.Undergrad     Outstate   Room.Board
Abilene Christian University 0.02454774445 0.2634297521 0.2395964691
Adelphi University           0.05614838562 0.5134297521 0.7361286255
                                    Books     Personal          PhD
Abilene Christian University 0.1577540107 0.2977099237 0.6526315789
Adelphi University           0.2914438503 0.1908396947 0.2210526316
                                 Terminal    S.F.Ratio perc.alumni
```

```
Abilene Christian University 0.71052631579 0.4182305630      0.1875
Adelphi University           0.07894736842 0.2600536193      0.2500
                                     Expend    Grad.Rate
Abilene Christian University 0.0726714046 0.4629629630
Adelphi University           0.1383867137 0.4259259259
```

Train and Test Split

Let us now split our data into a training set and a test set. We will run our neural network on the training set and then see how well it performed on the test set. We will use the caTools to randomly split the data into a training set and test set.

```
# Convert Private column from Yes/No to 1/0
Private = as.numeric(College$Private)-1
data = cbind(Private,scaled.data)

library(caTools)
set.seed(101)

# Create Split (any column is fine)
split = sample.split(data$Private, SplitRatio = 0.70)

# Split based off of split Boolean Vector
train = subset(data, split == TRUE)
test = subset(data, split == FALSE)
```

Neural Network Function

Before we actually call the neuralnetwork() function we need to create a formula to insert into the machine learning model. The neuralnetwork() function won't accept the typical decimal R format for a formula involving all features (e.g. y ~.). However, we can use a simple script to create the expanded formula and save us some typing:

```
feats <- names(scaled.data)

# Concatenate strings
f <- paste(feats,collapse=' + ')
f <- paste('Private ~',f)

# Convert to formula
f <- as.formula(f)

f
```

```
Private ~ Apps + Accept + Enroll + Top10perc + Top25perc + F.Undergrad +
    P.Undergrad + Outstate + Room.Board + Books + Personal +
    PhD + Terminal + S.F.Ratio + perc.alumni + Expend + Grad.Rate
```

```
#install.packages('neuralnet')
```

```
library(neuralnet)
nn <- neuralnet(f,train,hidden=c(10,10,10),linear.output=FALSE)
```

Predictions and Evaluations

Now let's see how well we performed! We use the compute() function with the test data (jsut the features) to create predicted values. This returns a list from which we can call net.result off of.

```
# Compute Predictions off Test Set
predicted.nn.values <- compute(nn,test[2:18])

# Check out net.result
print(head(predicted.nn.values$net.result))
```

```
                                                [,1]
Adrian College                           1.0000000000
Alfred University                        1.0000000000
Allegheny College                        1.0000000000
Allentown Coll. of St. Francis de Sales 0.9999999891
Alma College                             1.0000000000
Amherst College                          0.9999999994
...
```

Notice we still have results between 0 and 1 that are more like probabilities of belonging to each class. We'll use sapply() to round these off to either 0 or 1 class so we can evaluate them against the test labels.

```
predicted.nn.values$net.result <-
sapply(predicted.nn.values$net.result,round,digits=0)
```

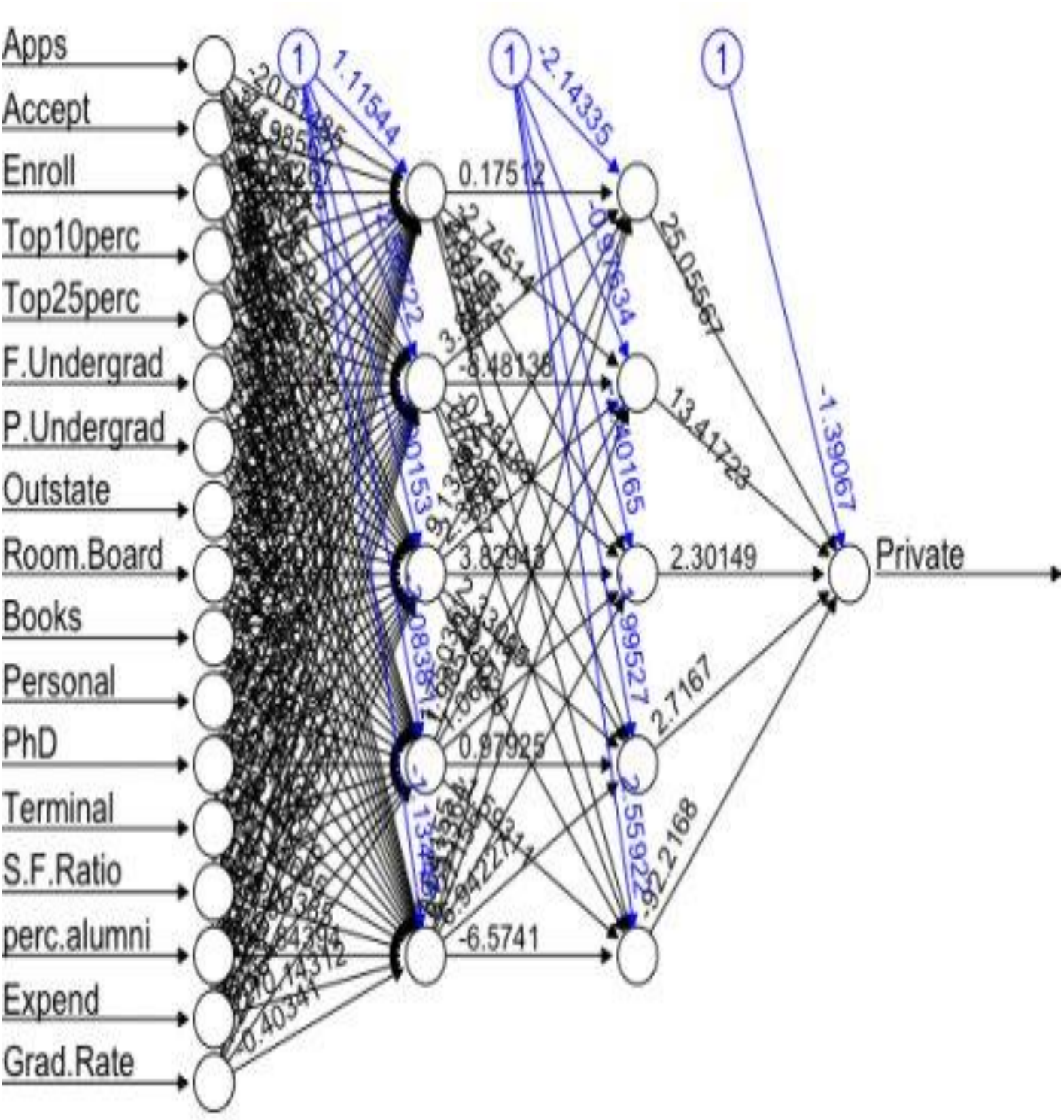
Now let's create a simple confusion matrix:

```
table(test$Private,predicted.nn.values$net.result)
```

```
     0   1
  0 57   7
  1  6 163
```

Visualizing the Neural Net

We can visualize the Neural Network by using the plot(nn) command. The black lines represent the weighted vectors between the neurons. The blue line represents the bias added. Unfortunately, even though the model is clearly a very powerful predictor, it is not easy to directly interpret the weights. This means that we usually have to treat Neural Network models more like black boxes.


Apps
Accept
Enroll
Top10perc
Top25perc
F.Undergrad
P.Undergrad
Outstate
Room.Board
Books
Personal
PhD
Terminal
S.F.Ratio
perc.alumni
Expend
Grad.Rate
1
1.11544
-2.14335
0.17512
-8.48138
3.82943
0.87925
-6.5741
25.05567
13.41722
2.30149
2.7167
-1.39067
Private