# Data Science Assignment

Sumeet Shivgand

December 4, 2019

# Question F

Develop a InfoGain algorithm that works on this dataset to calculate the variable for the first split. You may use the code developed in the labs as a starting point but make sure to annotate your code with comments explaining what it is doing. Note you can only used base R commands here no other packages are allowed. Comment on your results.

```
######Info-Gain######

set.seed(850)
#1. Use the readxl package to read the sheet from  "Credit_Risk6_final.xlsx" file.
sheet2<- read_excel("D:\\CIT\\Data Science and Analytics\\Assignment\\Credit_Risk6_final.xlsx",
 sheet="Training_Data")

#Generate the dataframe for above generated sheet.
InfoG <- data.frame(sheet2)
#View(InfoG)
#str(InfoG)

#Checking for missing values.
sum(is.na(InfoG))
```

```
## [1] 44
```

```
#copying original data
data.info<- InfoG

#Remove the NA values
data.info<- na.omit(data.info)

#Remove the 'ID' Column
data.info<- subset(data.info, select = Checking.Acct:Credit.Standing)
#View(df1)
sum(is.na(data.info))
```

```
## [1] 0
```

```
#Convert all character variables to Factors

cols1 <- c("Checking.Acct","Credit.History","Loan.Reason","Savings.Acct","Employment",
           "Personal.Status","Housing","Job.Type","Foreign.National","Credit.Standing" )
data.info %<>%
  mutate_each_(funs(factor(.)),cols1)
```

```
## Warning: mutate_each() is deprecated
## Please use mutate_if(), mutate_at(), or mutate_all() instead:
##
##   - To map `funs` over all variables, use mutate_all()
##   - To map `funs` over a selection of variables, use mutate_at()
## This warning is displayed once per session.
```

```
## Warning: funs() is soft deprecated as of dplyr 0.8.0
## Please use a list of either functions or lambdas:
##
##   # Simple named list:
##   list(mean = mean, median = median)
##
##   # Auto named with `tibble::lst()`:
##   tibble::lst(mean, median)
##
##   # Using lambdas
##   list(~ mean(., trim = .2), ~ median(., na.rm = TRUE))
## This warning is displayed once per session.
```

```
str(data.info)
```

```
## 'data.frame':    737 obs. of  13 variables:
##  $ Checking.Acct                       : Factor w/ 4 levels "0Balance","High",..: 4 1 1 4 3 1
3 4 3 1 ...
##  $ Credit.History                      : Factor w/ 5 levels "All Paid","Bank Paid",..: 1 4 4
1 4 1 2 5 3 4 ...
##  $ Loan.Reason                         : Factor w/ 10 levels "Business","Car New",..: 2 2 2 1
0 2 2 4 3 5 5 ...
##  $ Savings.Acct                        : Factor w/ 5 levels "High","Low","MedHigh",..: 2 2 5
5 4 2 2 2 2 2 ...
##  $ Employment                          : Factor w/ 6 levels "Long","Medium",..: 2 4 1 1 6 1 2
4 6 4 ...
##  $ Personal.Status                     : Factor w/ 3 levels "Divorced","Married",..: 3 1 1 3
1 2 1 2 1 3 ...
##  $ Housing                             : Factor w/ 3 levels "Other","Own",..: 2 2 2 1 2 2 1 3
3 2 ...
##  $ Job.Type                            : Factor w/ 4 levels "Management","Skilled",..: 1 2 2
2 4 2 4 2 2 2 ...
##  $ Foreign.National                    : Factor w/ 2 levels "No","Yes": 1 1 1 2 1 2 1 2 2 1
...
##  $ Months.since.Checking.Acct.opened   : num  7 16 25 7 13 22 25 13 13 19 ...
##  $ Residence.Time..In.current.district.: num  3 2 2 4 2 3 4 4 3 3 ...
##  $ Age                                 : num  44 28 28 35 22 29 33 40 24 41 ...
##  $ Credit.Standing                     : Factor w/ 2 levels "Bad","Good": 2 1 1 2 2 2 2 2 1 1
...
```

```
#Create a table and a proportional table - 1-D for Credit Standing

table(data.info$Credit.Standing)
```

```
##
##  Bad Good
##  305  432
```

```
creditpt <- prop.table(table(data.info$Credit.Standing))

# Now as before create a function so that we can create any 2-D table

cr.func <- function(x) {table(data.info[,x],data.info[,13])}
print(cr.func(2))  # checking does it work.
```

```
##
##             Bad Good
##   All Paid    5  125
##   Bank Paid  10   50
##   Critical   83    1
##   Current   183  219
##   Delay      24   37
```

```
# Create a function so that we can create any 2-D table
# Fill in z and run it.

cr.func <- function(x) {prop.table(table(data.info[,x],data.info[,13]), margin = 1)}
print(cr.func(2))  # checking does it work.
```

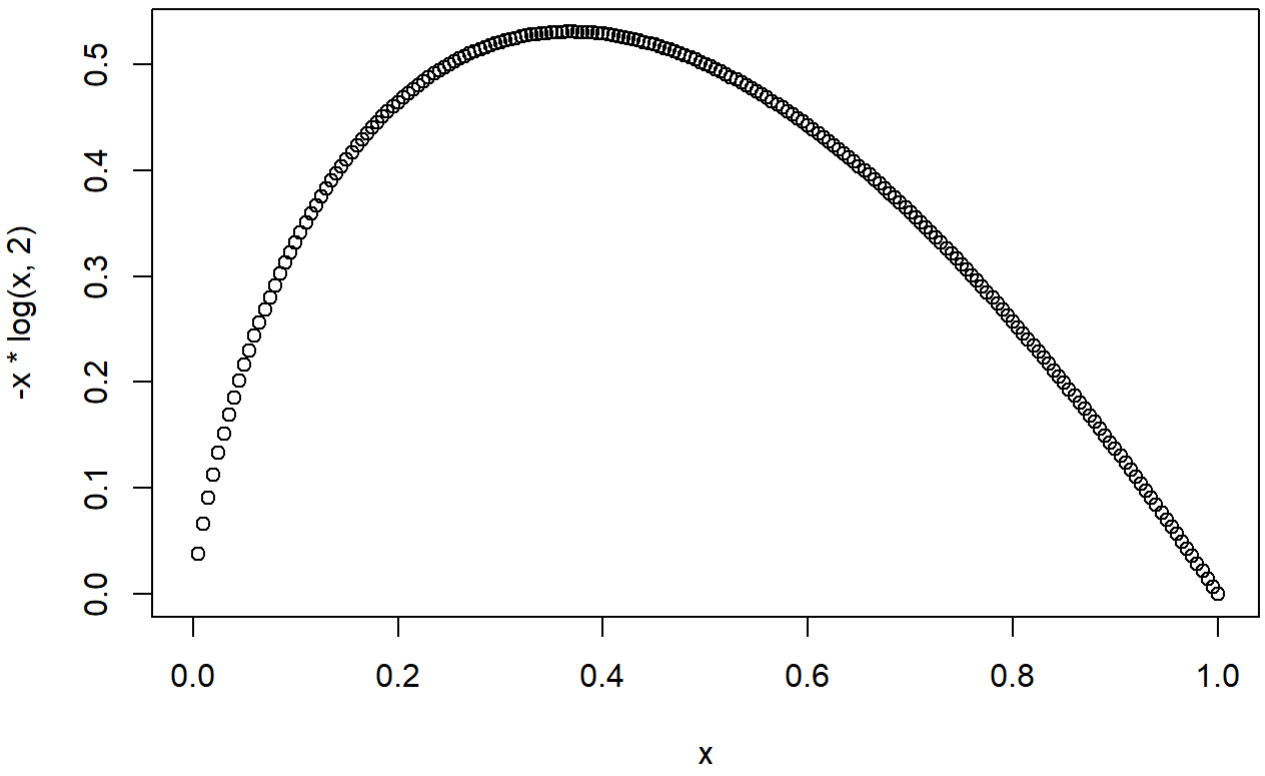
```
##
##                    Bad       Good
##   All Paid  0.03846154 0.96153846
##   Bank Paid 0.16666667 0.83333333
##   Critical  0.98809524 0.01190476
##   Current   0.45522388 0.54477612
##   Delay     0.39344262 0.60655738
```

```
# Now calculate the entropy - i.e. -1 *probability of a false * log2( of this probability)
-cr.func(2)*log2(cr.func(2))
```

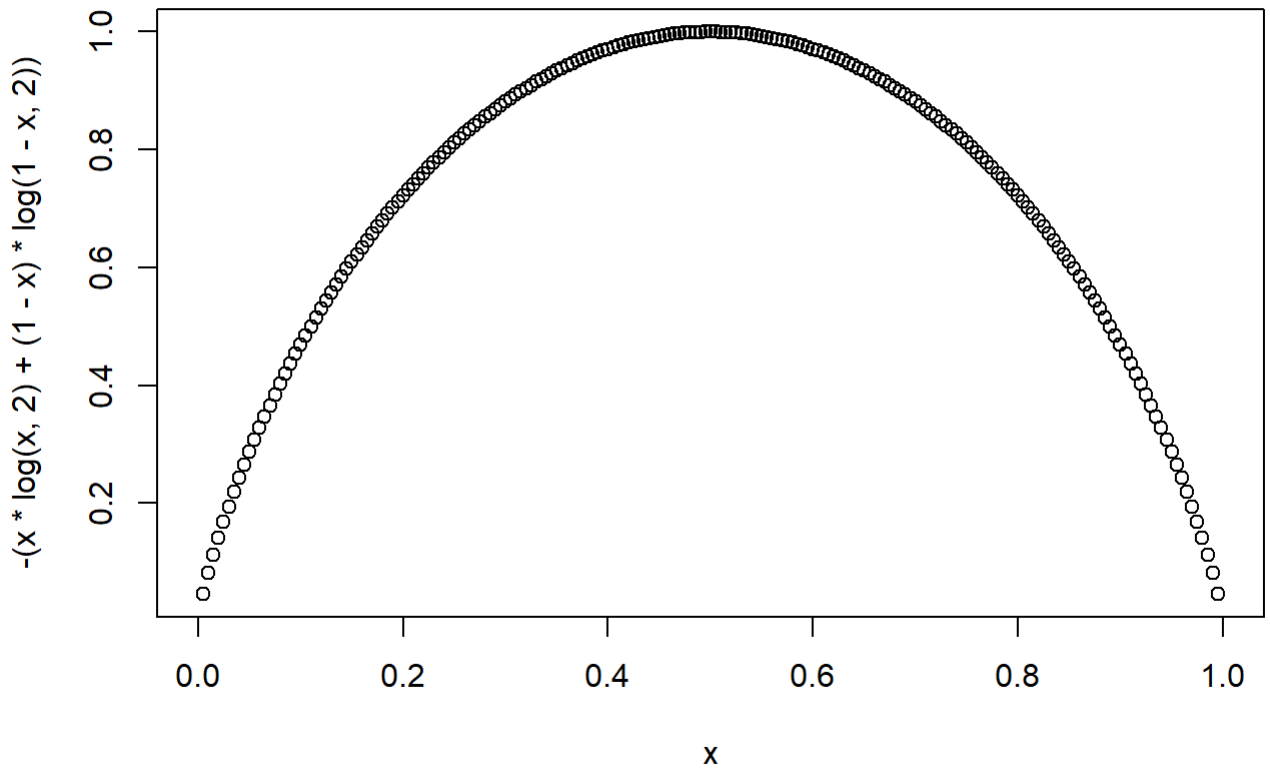
```
##
##                    Bad       Good
##   All Paid  0.18078614 0.05440724
##   Bank Paid 0.43082708 0.21919534
##   Critical  0.01707230 0.07609902
##   Current   0.51683928 0.47736805
##   Delay     0.52948518 0.43750011
```

```
# We can graph this for different probability values against entropy

x = seq(0,1, by = 0.005)
plot(x, -x*log(x,2))  # one class - but normally 2 or more classes
```




```
plot(x, -(x*log(x,2)+(1-x)*log(1-x,2))) # this is a 2-class problem
```




```
# Now we need rowSums of this, i.e the row sums

rowSums(-cr.func(2)*log2(cr.func(2)))
```

```
##   All Paid  Bank Paid   Critical    Current      Delay
## 0.23519338 0.65002242 0.09317132 0.99420732 0.96698530
```

```
# Next we need to multiply these by the proportion in each of these rows

prop.table(table(data.info$Credit.History))
```

```
##
##   All Paid  Bank Paid   Critical    Current      Delay
## 0.17639077 0.08141113 0.11397558 0.54545455 0.08276798
```

```
sum(prop.table(table(data.info$Credit.History))*rowSums(-cr.func(2)*log2(cr.func(2))))
```

```
## [1] 0.7273546
```

```
# Check that this is the same as the Excel calcs.

# Now bring it altogether with one formula

entopy_cr <- function(x) { cr.func <- prop.table(table(data.info[,x],data.info[,13]), margin = 1
)
sum(prop.table(table(data.info[,x]))*rowSums(-cr.func(x)*log2(cr.func(x)))) }

entopy_cr(2)
```

```
## [1] 0.7273546
```

```
# Check for other columns/variables in dataset
# Now create a for loop for other categorical variables of interest

r <- c(1,2,3,4,5,6,7,8,9)

r3 <- NULL
for (x in r) { r2 <- entopy_cr(x)
r3 <- c(r3,r2)
print(r3)
}
```

```
## [1] 0.9494604
## [1] 0.9494604 0.7273546
## [1] 0.9494604 0.7273546       NaN
## [1] 0.9494604 0.7273546       NaN 0.9724762
## [1] 0.9494604 0.7273546       NaN 0.9724762 0.9241720
## [1] 0.9494604 0.7273546       NaN 0.9724762 0.9241720 0.9722757
## [1] 0.9494604 0.7273546       NaN 0.9724762 0.9241720 0.9722757 0.9722452
## [1] 0.9494604 0.7273546       NaN 0.9724762 0.9241720 0.9722757 0.9722452
## [8] 0.9670490
## [1] 0.9494604 0.7273546       NaN 0.9724762 0.9241720 0.9722757 0.9722452
## [8] 0.9670490 0.9780362
```

```
#Now calculating information Gain:- (entropy of Parent - [Weighted avg]*entropy(Children Nodes)

info_tab <- (1-r3)
print(info_tab)
```

```
## [1] 0.05053958 0.27264542        NaN 0.02752378 0.07582796 0.02772428
## [7] 0.02775483 0.03295096 0.02196379
```

# Comment:-

Before starting with Information gain. We discuss about entropy. As we know decision tree is built top-down from root node and partition the data into subset that contain instances with similar values (homogeneous). ID3 algorithm uses entropy to calculate the homogeneity of a sample. If the sample is completely homogeneous the

entropy is zero and if the sample is equally divided then it has entropy of one. The information gain is based on the decrease in entropy after a data-set is split on an attribute. Building a decision tree is all about finding attribute that returns the highest information gain. Information gain is calculated by comparing the entropy of the dataset before and after a transformation. It can also be used for feature selection, by evaluating the gain of each variable in the context of the target variable. Now, I have find the information gain value for all the variables and I see that 'Credit History' variable has the highest information gain. So, the variable for first split will be 'Credit History'.

# Question G

Develop a working adabag using formulae in R. Generate a random prediction (each time) for 4 iterations of boosting. Include a confusion matrix at the end for your final prediction and comment. Only used base R commands here no other packages are allowed.

```
##Adaboost using Base R##

y=c(0, 1, 1, 0, 1, 1, 0, 1, 0, 0)

set.seed(850)
#Let t=1:
#Initializing weights to all training points
D1=rep(1/10,10)

#Generated random prediction for h1
h1<- round(runif(10, min=0, max=1))

#Calculating error rate for each classifier i.e. epsilon1
epsilon_1=sum(D1[!(y==h1)])
epsilon_1
```

```
## [1] 0.4
```

```
#Computing voting power for the classsifier
alpha_1=0.5*log((1-epsilon_1)/epsilon_1)
alpha_1
```

```
## [1] 0.2027326
```

```
#Append the classifier in the ensemble classifier D2
D2=c()

#Update weights of each points where the previous classifier went wrong
for (i in seq(1:length(y))){
  if (h1[i]==y[i]) {
    F[i]=exp(-alpha_1)
    } else {F[i]=exp(alpha_1)}
  D2[i]=D1[i]*F[i]
}
F
```

```
##  [1] 1.2247449 0.8164966 0.8164966 1.2247449 0.8164966 1.2247449 0.8164966
##  [8] 1.2247449 0.8164966 0.8164966
```

```
###############Let t=2:#################
#Initializing weights to all training points for iteration 2
D2=D2/sum(D2)
D2
```

```
##  [1] 0.12500000 0.08333333 0.08333333 0.12500000 0.08333333 0.12500000
##  [7] 0.08333333 0.12500000 0.08333333 0.08333333
```

```
#Generated random prediction for h2
h2<- round(runif(10, min=0, max=1))

#Calculating error rate for each classifier i.e. epsilon2
epsilon_2=sum(D2[!(y==h2)])
epsilon_2
```

```
## [1] 0.625
```

```
#Computing voting power for the classsifier
alpha_2=0.5*log((1-epsilon_2)/epsilon_2)
alpha_2
```

```
## [1] -0.2554128
```

```
#Append the classifier in the ensemble classifier D3
D3=c()

#Update weights of each points where the previous classifier went wrong
for (i in seq(1:length(y))){
  if (h2[i]==y[i]) {
    F[i]=exp(-alpha_2)
    } else {F[i]=exp(alpha_2)}
  D3[i]=D2[i]*F[i]
}
F
```

```
##  [1] 0.7745967 1.2909944 0.7745967 1.2909944 0.7745967 0.7745967 1.2909944
##  [8] 0.7745967 0.7745967 1.2909944
```

```
#################Let t=3:################
#Initializing weights to all training points for iteration 3
D3=D3/sum(D3)
D3
```

```
##  [1] 0.10000000 0.11111111 0.06666667 0.16666667 0.06666667 0.10000000
##  [7] 0.11111111 0.10000000 0.06666667 0.11111111
```

```
#Generated random prediction for h3
h3<- round(runif(10, min=0, max=1))

#Calculating error rate for each classifier i.e. epsilon3
epsilon_3=sum(D3[!(y==h3)])
epsilon_3
```

```
## [1] 0.8222222
```

```
#Computing voting power for the classsifier
alpha_3=0.5*log((1-epsilon_3)/epsilon_3)
alpha_3
```

```
## [1] -0.7657382
```

```
#Append the classifier in the ensemble classifier D4
D4=c()

#Update weights of each points where the previous classifier went wrong
for (i in seq(1:length(y))){
  if (h3[i]==y[i]) {
    F[i]=exp(-alpha_3)
    } else {F[i]=exp(alpha_3)}
  D4[i]=D3[i]*F[i]
}
F
```

```
##  [1] 0.4649906 0.4649906 2.1505813 0.4649906 0.4649906 0.4649906 2.1505813
##  [8] 0.4649906 0.4649906 0.4649906
```

```
##################Let t=4:#################
#Initializing weights to all training points for iteration 4
D4=D4/sum(D4)
D4
```

```
##  [1] 0.06081081 0.06756757 0.18750000 0.10135135 0.04054054 0.06081081
##  [7] 0.31250000 0.06081081 0.04054054 0.06756757
```

```
#Generated random prediction for h4
h4<- round(runif(10, min=0, max=1))

#Calculating error rate for each classifier i.e. epsilon4
epsilon_4=sum(D4[!(y==h4)])
epsilon_4
```

```
## [1] 0.3581081
```

```
#Computing voting power for the classsifier
alpha_4=0.5*log((1-epsilon_4)/epsilon_4)
alpha_4
```

```
## [1] 0.2917925
```

```
h_final=sign(alpha_1*h1+alpha_2*h2+alpha_3*h3+alpha_4*h4)
h_final
```

```
##  [1] -1 -1 -1 -1  1  0  0  0 -1 -1
```

```
#Notice how accurate this is:
h_final==y
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
```

```
table(h_final,y)
```

```
##        y
## h_final 0 1
##      -1 4 2
##      0  1 2
##      1  0 1
```

```
a <- vector(mode = "numeric", 4)
H_x<- t(4 * t(h_final))
p<- sign(rowSums(H_x))

eval_model(p,y)
```

```
##
## Confusion matrix (absolute):
##           Actual
## Prediction -1  0  1 Sum
##        -1   0  4  2   6
##        0    0  1  2   3
##        1    0  0  1   1
##        Sum  0  5  5  10
##
## Confusion matrix (relative):
##           Actual
## Prediction  -1   0   1 Sum
##        -1  0.0 0.4 0.2 0.6
##        0   0.0 0.1 0.2 0.3
##        1   0.0 0.0 0.1 0.1
##        Sum 0.0 0.5 0.5 1.0
##
## Accuracy:
## 0.2 (2/10)
##
## Error rate:
## 0.8 (8/10)
##
## Error rate reduction (vs. base rate):
## -0.6 (p-value = 0.9893)
```

# Comment:-

Let start with boosting. Boosting algorithm converts the weak learners into strong learners. Boosting is a sequential process i.e. trees are grown using the information from previously grown tree one after the other. This process learns slowly from data and tries to improve its prediction in subsequent iterations. Now we see how adaboost works. Steps are as follows:-

1. Initialize the weights to all training points(D=1/n)
2. Calculate the error rate for each weak classifier
3. Pick the classifier with the lowest error rate
4. Then, compute the voting power for the classifier. This means it gives the weak classifiers strength in terms of how much is it more accurate as compared to other week classifiers
5. Append the classifier in the ensemble classifier
6. Update weights of each points where the previous classifier went wrong.

At the end, two confusion matrix is generated, absolute and relative. Then, we get 20% or 0.2 accuracy(2/10) and 80% or 0.8 error rate(8/10).

# Question H

Generate ROC curve using base R code

```
#Use the readxl package to read the sheet from  "Credit_Risk6_final.xlsx" file.
sheet1<- read_excel("D:\\CIT\\Data Science and Analytics\\Assignment\\Credit_Risk6_final.xlsx",
 sheet="Training_Data")

#Generate the dataframe for above generated sheet.
roc_df <- data.frame(sheet1)
#View(df)
#str(roc_df)

#Checking for missing values.
sum(is.na(roc_df))
```

```
## [1] 44
```

```
#copying original data
roc_df1<- roc_df

#Remove the NA values
roc_df1<- na.omit(roc_df1)

#Remove the 'ID' Column
roc_df1<- subset(roc_df1, select = Checking.Acct:Credit.Standing)
#View(df1)
sum(is.na(roc_df1))
```

```
## [1] 0
```

```
#Convert all character variables to Factors

cols1 <- c("Checking.Acct","Credit.History","Loan.Reason","Savings.Acct","Employment",
           "Personal.Status","Housing","Job.Type","Foreign.National","Credit.Standing")
roc_df1 %<>%
  mutate_each_(funs(factor(.)),cols1)
#str(roc_df1)

#Copying the values of 'Credit Standing' variable of dataset to "Credit.Standing"
Credit.Standing <- roc_df1$Credit.Standing
#######ROC Curve###########
set.seed(850)
rf <- randomForest(Credit.Standing~.,data=roc_df1)

# Model output
print(rf)
```

```
##
## Call:
##  randomForest(formula = Credit.Standing ~ ., data = roc_df1)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 25.37%
## Confusion matrix:
##      Bad Good class.error
## Bad  210   95   0.3114754
## Good  92  340   0.2129630
```

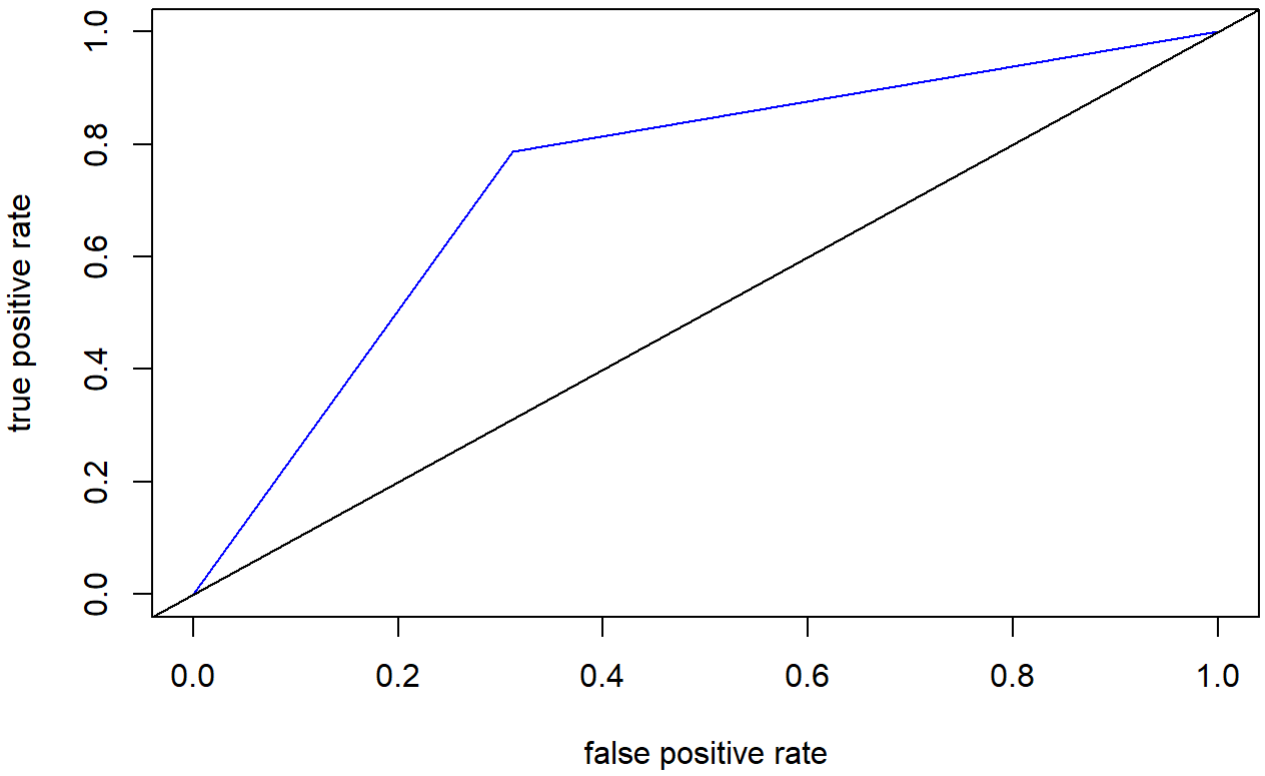
```
df1.ypos <- as.numeric(rf$pred[Credit.Standing == "Good"]) -1
df1.yneg <- as.numeric(rf$pred[Credit.Standing == "Bad"]) - 1

sort.df1.y <- as.numeric(sort(rf$pred)) - 1

sens <- 0
spec <- 0

for (i in length(sort.df1.y):1){
  sens <- (c(sens, mean(df1.ypos >= sort.df1.y[i])))
  spec <- (c(spec, mean(df1.yneg >= sort.df1.y[i])))
}

plot(spec, sens, xlim = c(0, 1), ylim = c(0, 1), type = "l",
     xlab = "false positive rate", ylab = "true positive rate", col = 'blue')
abline(0, 1, col= "black")
```




```
#Separating Positive and Negative as Good and Bad
positive<-sum(sort.df1.y)
negative<- sum(!sort.df1.y)

#True Positive Rate
TPR<- positive/(positive+negative)
print(TPR)
```

```
## [1] 0.5902307
```

```
#False Negative Rate
FNR<- negative/(positive+negative)
print(FNR)
```

```
## [1] 0.4097693
```

# Comment:-

ROC curve is basically plot of ‘Sensitivity’ versus ‘1 - Specificity’. Sensitivity is nothing but positive recall. It means out off all positive samples, how many samples classifier able to pickup. Sensitivity is also called ‘True Positive Rate’[TP/(TP+FN)]. Whereas, Specificity is nothing but negative recall. It means out off all negative samples, how

many samples classifier able to pickup. Specificity is also called ‘False Negative Rate’[TN/(TN+FP)]. [1 – Specificity= FP/(TN+FP)]. If the probability ratio is greater than threshold value then it will consider as 1 else it will be 0.