PM_final_project

#PART 1 - LOGISTIC REGRESSION AND NAIVE BAYES

```
### Library Imports ###
library(glmnet)
```

```
## Loading required package: Matrix

## Loading required package: foreach

## Loaded glmnet 2.0-18
```

```
library(caret) #for confusion matrix
```

```
## Loading required package: lattice

## Loading required package: ggplot2
```

```
library(e1071)
library(data.table)

### DATA LOADING , TEST/TRAIN SPLIT####

raw=read.csv("cleansed_data.csv")
raw = na.omit(raw)

raw=raw[,-1] #removing index column

#Converting to factors
factor_vars1 = c(2, 3, 4, 24)
for (i in factor_vars1) {
  raw[[i]]<-as.factor(raw[[i]])
}

set.seed(1)
train = sample(1:nrow(raw),(0.8*nrow(raw)))
test = (-train)

train.data <- raw[train,]
test.data <- raw[test,]


### VARIABLE IMPORTANCE USING LASSO CV ###

x=model.matrix(def_pay . + SEX*MARRIAGE + SEX*EDUCATION + MARRIAGE,raw)[,-1] ## Adding interaction terms
y=raw$def_pay

var_names = names(x[1,])
var_count = rep(0, length(var_names))
```

```r
for(i in c(1:10)){
  train = sample(1: nrow(x), (0.80)*nrow(x))
  test=(-train)
  cvfit.lasso = cv.glmnet(x[train,],y[train],
                          alpha=1, family = "binomial", standardize = TRUE)

  lasso_coef = predict(cvfit.lasso, newx=x[test,],
                       s = "lambda.min", type="coefficients")
  var_used_index = lasso_coef[-2]
  var_used_index = ifelse(var_used_index ==0, FALSE, TRUE)
  var_count[var_used_index] = var_count[var_used_index] +1
  print(paste0("done", i))
}
```

```
## [1] "done1"
## [1] "done2"
## [1] "done3"
## [1] "done4"
## [1] "done5"
## [1] "done6"
## [1] "done7"
## [1] "done8"
## [1] "done9"
## [1] "done10"
```

```r
var_used_index_n = ifelse(var_count == 10, TRUE, FALSE)
names(data.frame(x))[var_used_index_n]
```

```
##  [1] "LIMIT_BAL." "SEX2"       "EDUCATION3" "MARRIAGE2"  "AGE"
##  [6] "PAY_1"      "PAY_2"      "PAY_3"      "PAY_4"      "PAY_5"
## [11] "PAY_6"      "BILL_AMT1." "PAY_AMT1."  "PAY_AMT2."  "PAY_AMT4."
## [16] "PAY_AMT6."
```

```r
### LOGISTIC REGRESSION ###
logit.model <- glm(def_pay~ LIMIT_BAL.+SEX+MARRIAGE+PAY_1+PAY_2+
                     PAY_3+PAY_4+PAY_5+PAY_6+BILL_AMT1.+PAY_AMT1.+PAY_AMT2.+PAY_AMT4.+PAY_AMT6.
                   ,data=train.data, family = binomial(link = "logit"))
logit.prob <- predict(logit.model,test.data,type = 'response')

threshold.logit <- 0.50
pred.logit <- ifelse(logit.prob > threshold.logit,1,0)

confusionMatrix(table(pred.logit, test.data$def_pay))
```

```
## Confusion Matrix and Statistics
##
##
## pred.logit    0    1
##          0 4346  843
##          1  149  334
##
```

```
##                Accuracy : 0.8251
##                  95% CI : (0.815, 0.8349)
##     No Information Rate : 0.7925
##     P-Value [Acc > NIR] : 3.549e-10
##
##                   Kappa : 0.3203
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9669
##             Specificity : 0.2838
##          Pos Pred Value : 0.8375
##          Neg Pred Value : 0.6915
##              Prevalence : 0.7925
##          Detection Rate : 0.7662
##    Detection Prevalence : 0.9148
##       Balanced Accuracy : 0.6253
##
##        'Positive' Class : 0
##
```

```
### NAIVE BAYES MODEL ###
naive.model = naiveBayes(def_pay~LIMIT_BAL.+SEX+MARRIAGE+PAY_1+PAY_2+PAY_3+PAY_4+PAY_5+PAY_6+BILL_AMT1.
                         +PAY_AMT1.+PAY_AMT2.+PAY_AMT4.+PAY_AMT6., data=train.data, type="raw")

naive.prob <- predict(naive.model, test.data, type='raw')
naive.prob<-data.table((naive.prob))

threshold.naive <- 0.50
pred.naive <- ifelse(naive.prob$`1` > threshold.naive,1,0)

confusionMatrix(table(pred.naive, test.data$def_pay))
```

```
## Confusion Matrix and Statistics
##
##
## pred.naive    0    1
##          0 3748  467
##          1  747  710
##
##                Accuracy : 0.786
##                  95% CI : (0.7751, 0.7966)
##     No Information Rate : 0.7925
##     P-Value [Acc > NIR] : 0.89
##
##                   Kappa : 0.4018
##
##  Mcnemar's Test P-Value : 1.171e-15
##
##             Sensitivity : 0.8338
##             Specificity : 0.6032
##          Pos Pred Value : 0.8892
##          Neg Pred Value : 0.4873
##              Prevalence : 0.7925
```

```
##          Detection Rate : 0.6608
##    Detection Prevalence : 0.7431
##       Balanced Accuracy : 0.7185
##
##        'Positive' Class : 0
##
```

#PART 2 - KNN CLASSIFICATION

KNN CLASSIFICATION WITHOUT NORMALIZATION

Reading the data

```
rm(list=ls())
library(tinytex)
credit_default1=read.csv('cleansed_data.csv',header=T)
credit_default1=credit_default1[-1]
names(credit_default1)
```

```
##  [1] "LIMIT_BAL." "SEX"        "EDUCATION"  "MARRIAGE"   "AGE"
##  [6] "PAY_1"      "PAY_2"      "PAY_3"      "PAY_4"      "PAY_5"
## [11] "PAY_6"      "BILL_AMT1." "BILL_AMT2." "BILL_AMT3." "BILL_AMT4."
## [16] "BILL_AMT5." "BILL_AMT6." "PAY_AMT1."  "PAY_AMT2."  "PAY_AMT3."
## [21] "PAY_AMT4."  "PAY_AMT5."  "PAY_AMT6."  "def_pay"
```

Converting categorical variables

```
factor_vars1 = c(2, 3, 4, 24, c(6:11))
for (i in factor_vars1) {
credit_default1[[i]]<-as.factor(credit_default1[[i]])
}
```

Splitting the data

```
library(kknn)
```

```
##
## Attaching package: 'kknn'

## The following object is masked from 'package:caret':
##
##     contr.dummy
```

```r
library(class)
set.seed(33)
tr1 = sample(c(1:dim(credit_default1)[1]), 20000)
train1 = credit_default1[tr1,]
test1 = credit_default1[-tr1,]
y_train1=credit_default1[tr1,24]
y_test1=credit_default1[-tr1,24]
```

KNN using out of sample predictions for k=50

```r
set.seed(33)
library(caret)
library(precrec)
```

```
##
## Attaching package: 'precrec'

## The following object is masked from 'package:glmnet':
##
##     auc
```

```r
library(ROCit)
library(plotROC)
library(ggplot2)

knn1=knn(train1,test1,cl=y_train1,k=50,prob=FALSE,use.all=FALSE)
knn1_table=table(knn1,y_test1)
accuracy <- function(x){sum(diag(x)/(sum(rowSums(x)))) * 100}

x=accuracy(knn1_table)
precision=posPredValue(knn1,y_test1,positive='1')
recall=sensitivity(knn1,y_test1,positive = '1')
F1=(2*precision*recall)/(precision+recall)

print(knn1_table)
```

```
##     y_test1
## knn1    0    1
##    0 6505 1615
##    1  129  111
```

```r
cat("Accuracy is",x,"and misclassification rate is",100-x)
```

```
## Accuracy is 79.13876 and misclassification rate is 20.86124
```

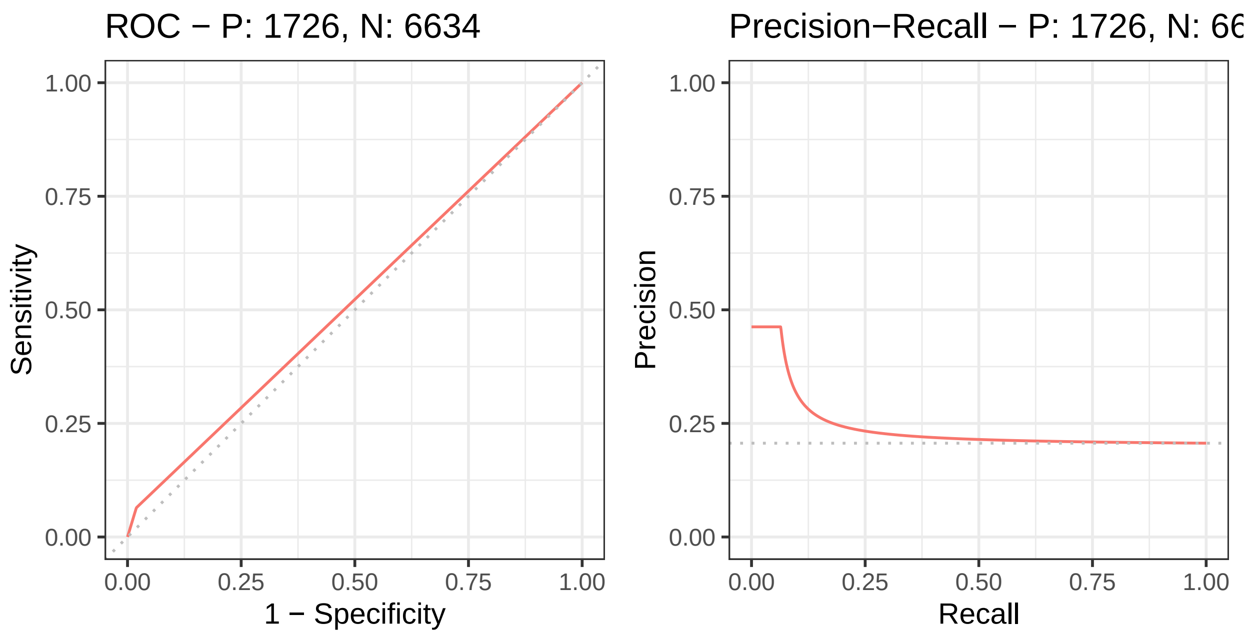
```r
cat("Precision is",precision,"and Recall is",recall)
```

```
## Precision is 0.4625 and Recall is 0.06431054
```

```r
cat("F1 score is",F1)
```

```
## F1 score is 0.1129196
```

```r
precrec_obj <- evalmod(scores = as.numeric(knn1), labels = y_test1)
autoplot(precrec_obj)
```

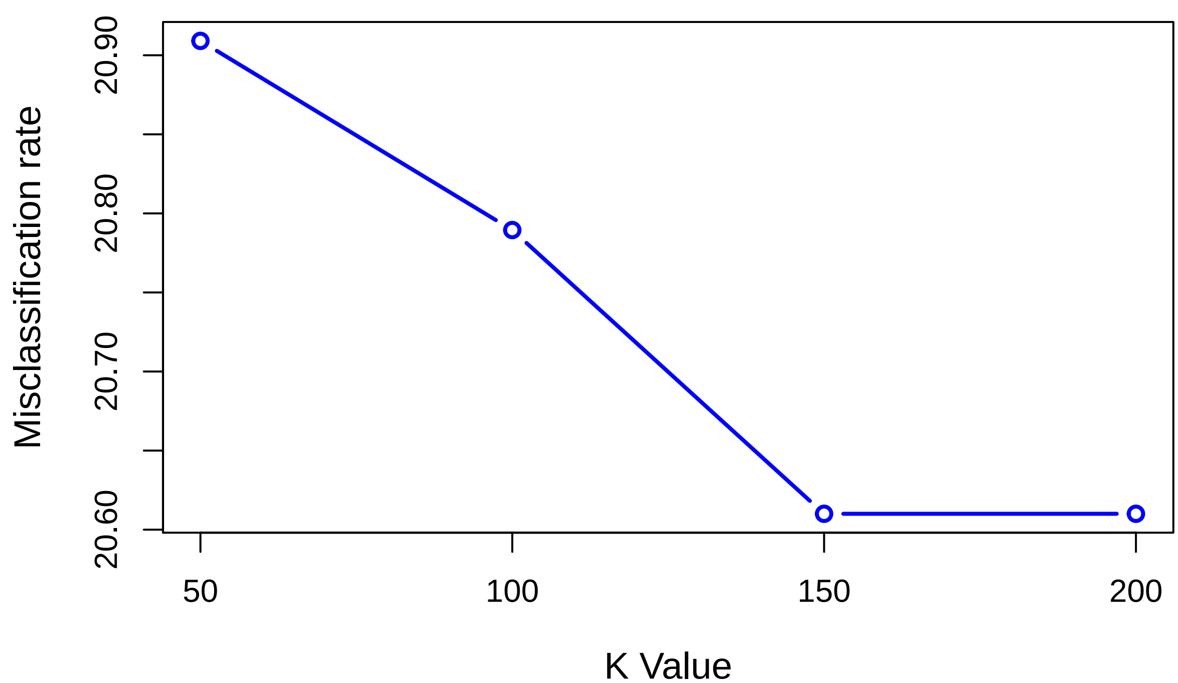


Finding the best k value

```r
set.seed(33)
tr1 = sample(c(1:dim(credit_default1)[1]), 20000)
train1 = credit_default1[tr1,]
test1 = credit_default1[-tr1,]
y_train1=credit_default1[tr1,24]
y_test1=credit_default1[-tr1,24]
k_list1=c(50,100,150,200)
k_list1
```

```
## [1]  50 100 150 200
```

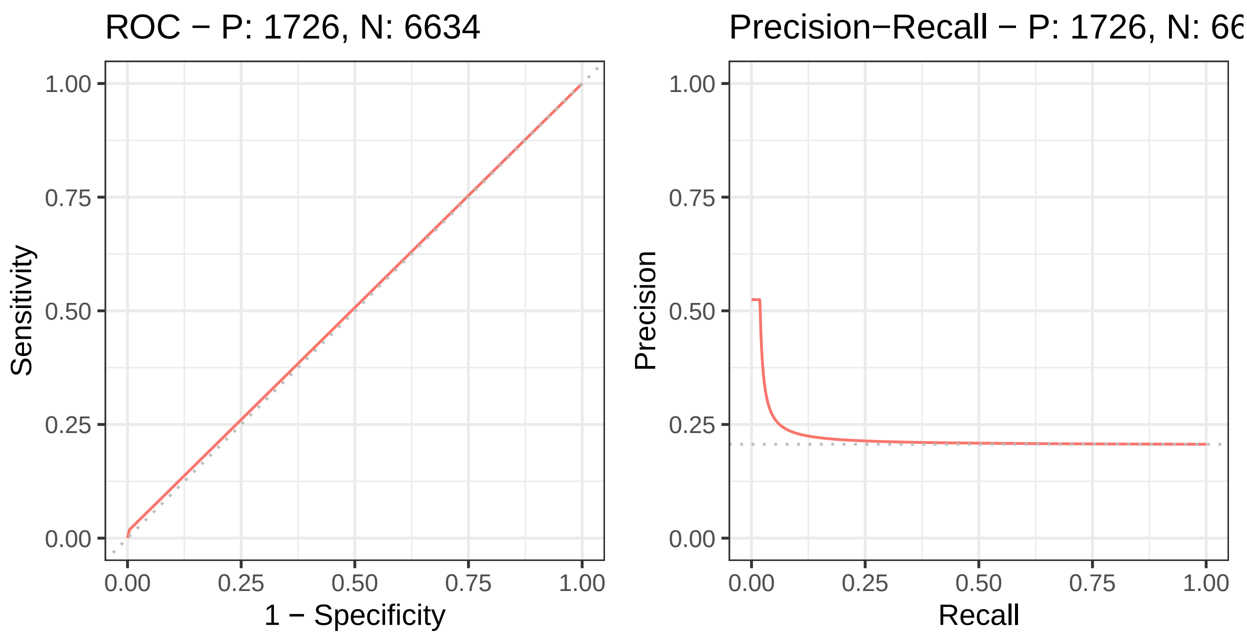
```
y=NULL
for(i in k_list1){
  knn2 = knn(train1,test1,cl=y_train1,k=i)
  knn2_table=table(knn2,y_test1)

  x=accuracy(knn2_table)
  precision=posPredValue(knn2,y_test1,positive='1')
  recall=sensitivity(knn2,y_test1,positive = '1')
  F1=(2*precision*recall)/(precision+recall)

  x1=100-x
  y=c(y,x1)
}
best = which.min(y)
plot(k_list1,y,type="b",xlab="K Value",col="blue",ylab="Misclassification rate",lwd=2,cex.lab=1.2)
```




```
cat("Best k value is",k_list1[best],"with misclassification rate of",y[best])
```

```
## Best k value is 200 with misclassification rate of 20.61005
```

```
precrec_obj <- evalmod(scores = as.numeric(knn2), labels = y_test1)
autoplot(precrec_obj)
```



K fold cross validation with kcv=10

```
set.seed(33)
train1 = credit_default1
test1 = credit_default1
y_train1=credit_default1[tr1,24]
y_test1=credit_default1[-tr1,24]
n=dim(credit_default1)[1]
k_list1=c(20,50,100,150,200,250)
kcv = 10
n0 = round(n/kcv,0)
set=1:n
used = NULL
y1=matrix(0,kcv,6)
y2=matrix(0,kcv,6)
y3=matrix(0,kcv,6)
for(j in 1:kcv){
  if(n0<length(set)){val = sample(set,n0)}
  if(n0>=length(set)){val=set}
    train_i = train1[-val,]
    test_i = test1[val,]
    y_train_i=credit_default1[-val,24]
    y_test_i=credit_default1[val,24]
    for(i in 1:6){
```

```
    knn3 = knn(train_i,test_i,cl=y_train_i,k=k_list1[i])
    knn3_table=table(knn3,y_test_i)
    x1=accuracy(knn3_table)
    x2=100-x1
    precision=posPredValue(knn3,y_test_i,positive='1')
    recall=sensitivity(knn3,y_test_i,positive = '1')
    F1=(2*precision*recall)/(precision+recall)

    #cat(x2,"for k value",k_list1[i],"and fold",j,"\n")
    y1[j,i]=x2
    y2[j,i]=precision
    y3[j,i]=recall
   }
  used = union(used,val)
  set = (1:n)[-used]
  cat(j,'\n')
}
```

```
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 9
## 10
```

```
my1=apply(y1,2,mean)
my2=apply(y2,2,mean)
my3=apply(y3,2,mean)
cat("Misclassification rate values:",my1)
```

```
## Misclassification rate values: 21.43159 21.2165 21.1354 21.10719 21.15656 21.12482
```

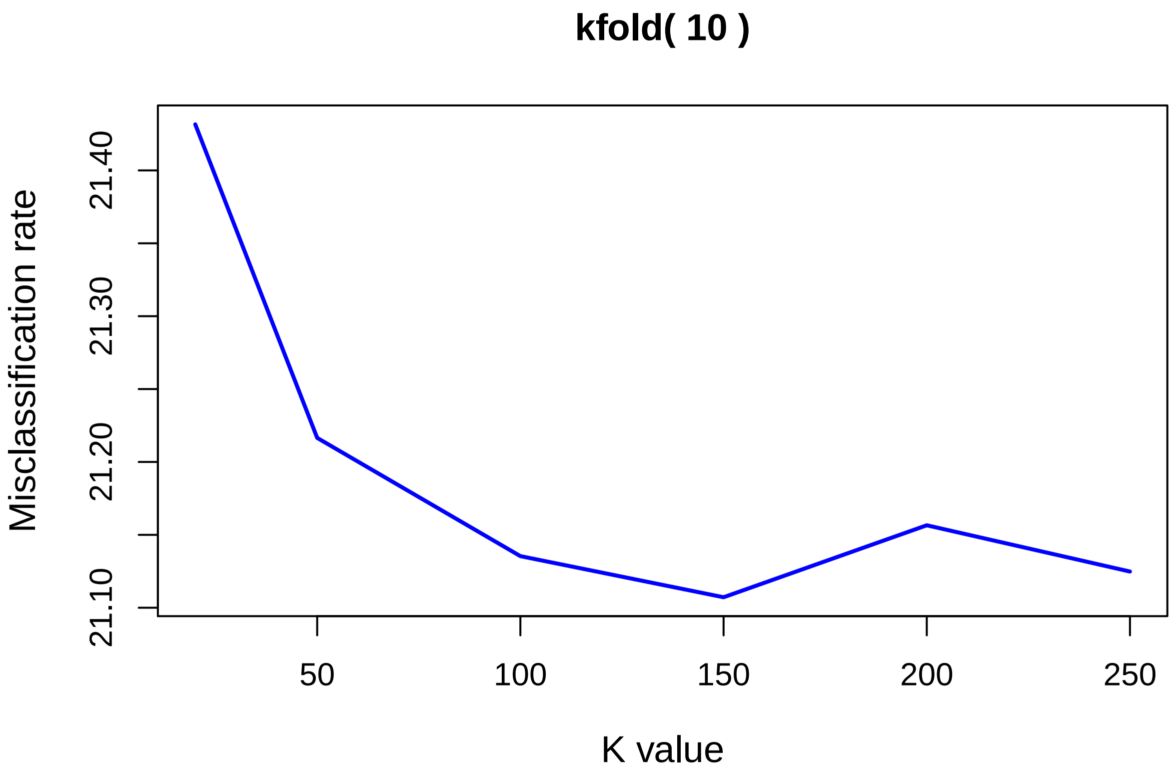
```
cat("Precision values:",my2)
```

```
## Precision values: 0.4683588 0.4938705 0.5114631 0.5266601 0.5233465 0.5652564
```

```
cat("Recall values:",my3)
```

```
## Recall values: 0.09199077 0.06881102 0.05439242 0.0381124 0.02404008 0.01219813
```

```
best = which.min(my1)
plot(k_list1,my1,xlab="K value",ylab="Misclassification rate",col=4,lwd=2,type="l",cex.lab=1.2,main=pas
```




```
cat("Best k value is",k_list1[best],"with misclassification rate of",my1[best])
```

```
## Best k value is 150 with misclassification rate of 21.10719
```

K fold cross validation with kcv=5

```
set.seed(33)
train1 = credit_default1
test1 = credit_default1
y_train1=credit_default1[tr1,24]
y_test1=credit_default1[-tr1,24]
n=dim(credit_default1)[1]
k_list1=c(20,50,100,150,200,250)
kcv = 5
n0 = round(n/kcv,0)
set=1:n
used = NULL
y1=matrix(0,kcv,6)
y2=matrix(0,kcv,6)
y3=matrix(0,kcv,6)
for(j in 1:kcv){
  if(n0<length(set)){val = sample(set,n0)}
  if(n0>=length(set)){val=set}
```

```r
    train_i = train1[-val,]
    test_i = test1[val,]
    y_train_i=credit_default1[-val,24]
    y_test_i=credit_default1[val,24]
    for(i in 1:6){
     knn4 = knn(train_i,test_i,cl=y_train_i,k=k_list1[i])
     knn4_table=table(knn4,y_test_i)
     x1=accuracy(knn4_table)
     x2=100-x1
     precision=posPredValue(knn4,y_test_i,positive='1')
     recall=sensitivity(knn4,y_test_i,positive = '1')
     F1=(2*precision*recall)/(precision+recall)

     #cat(x2,"for k value",k_list1[i],"and fold",j,"\n")
     y1[j,i]=x2
     y2[j,i]=precision
     y3[j,i]=recall
    }
  used = union(used,val)
  set = (1:n)[-used]
  cat(j,'\n')
}
```

```
## 1
## 2
## 3
## 4
## 5
```

```r
my1=apply(y1,2,mean)
my2=apply(y2,2,mean)
my3=apply(y3,2,mean)
cat("Misclassification rate values:",my1)
```

```
## Misclassification rate values: 21.41749 21.05078 21.04372 21.01551 21.14598 21.14951
```

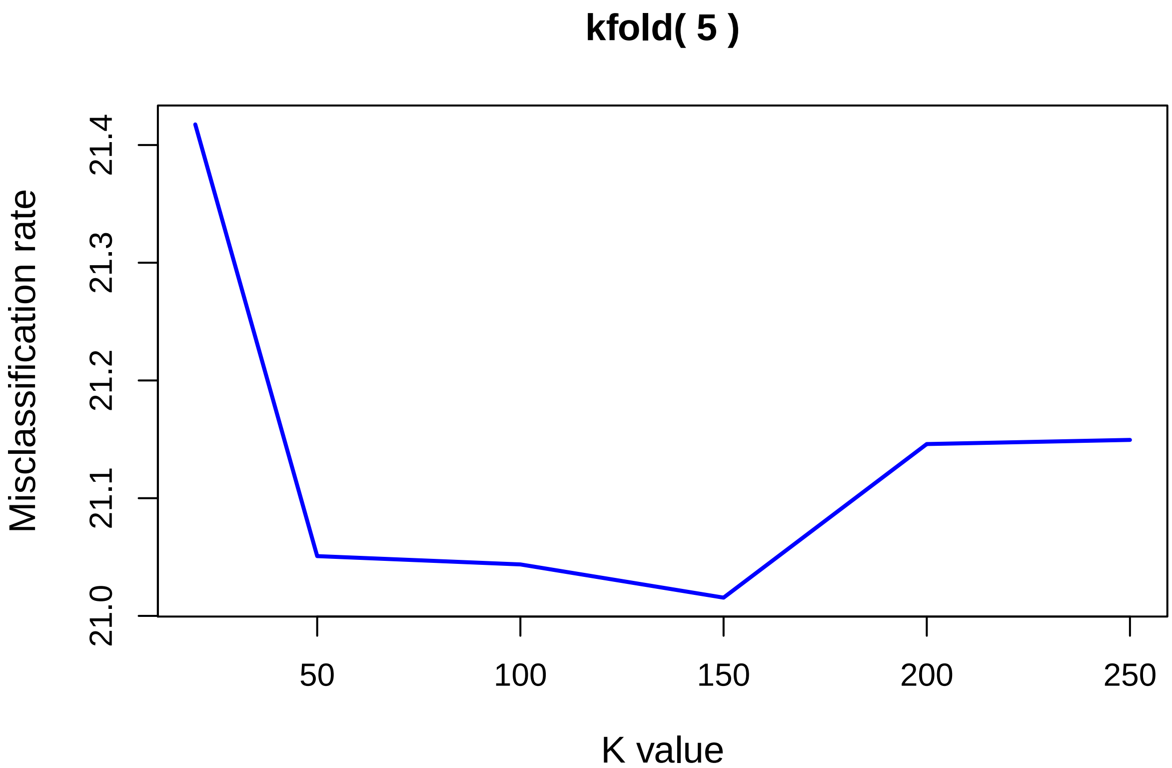
```r
cat("Precision values:",my2)
```

```
## Precision values: 0.4699687 0.5201941 0.5350113 0.5631058 0.5147846 0.5289855
```

```r
cat("Recall values:",my3)
```

```
## Recall values: 0.09271139 0.07085947 0.05050885 0.03643094 0.01864195 0.009495306
```

```r
best = which.min(my1)
plot(k_list1,my1,xlab="K value",ylab="Misclassification rate",col=4,lwd=2,type="l",cex.lab=1.2,main=pas
```




```r
cat("Best k value is",k_list1[best],"with misclassification rate of",my1[best])
```

```
## Best k value is 150 with misclassification rate of 21.01551
```

KNN CLASSIFICATION WITH NORMALIZATION

Reading the data

```r
rm(list=ls())
credit_default2=read.csv('cleansed_data.csv',header=T)
credit_default2=credit_default2[-1]

dim(credit_default2)
```

```
## [1] 28360    24
```

Converting categorical variables

```r
factor_vars2 = c(2, 3, 4, 24, c(6:11))
for (i in factor_vars2) {
credit_default2[[i]]<-as.factor(credit_default2[[i]])
}
```

Normalization

```r
normalize <- function(x) {
return ((x - min(x)) / (max(x) - min(x))) }
normalize_vars=c(1,5,c(12:23))
for (i in normalize_vars) {
credit_default2[[i]]=normalize(credit_default2[[i]])
}
summary(credit_default2)
```

```
##    LIMIT_BAL.       SEX       EDUCATION  MARRIAGE       AGE
##  Min.   :0.00000   1:11207   1:10216   1:12887   Min.   :0.0000
##  1st Qu.:0.05063   2:17153   2:13607   2:15473   1st Qu.:0.1892
##  Median :0.16456             3: 4537             Median :0.3514
##  Mean   :0.19875                                 Mean   :0.3777
##  3rd Qu.:0.29114                                 3rd Qu.:0.5405
##  Max.   :1.00000                                 Max.   :1.0000
##
##      PAY_1           PAY_2           PAY_3           PAY_4
##  0      :14082   0      :15035   0      :15056   0      :15725
##  -1     : 5504   -1     : 5807   -1     : 5687   -1     : 5439
##  1      : 3281   2      : 3794   2      : 3697   -2     : 3808
##  2      : 2560   -2     : 3231   -2     : 3546   2      : 3053
##  -2     : 2492   3      :  318   3      :  227   3      :  172
##  3      :  306   4      :   92   4      :   72   4      :   66
##  (Other):  135   (Other):   83   (Other):   75   (Other):   97
##      PAY_5           PAY_6         BILL_AMT1.       BILL_AMT2.
##  0      :16182   0      :15574   Min.   :0.0000   Min.   :0.00000
##  -1     : 5296   -1     : 5482   1st Qu.:0.1859   1st Qu.:0.08995
##  -2     : 4004   -2     : 4324   Median :0.2070   Median :0.11288
##  2      : 2550   2      : 2681   Mean   :0.2380   Mean   :0.14663
##  3      :  173   3      :  179   3rd Qu.:0.2557   3rd Qu.:0.16516
##  4      :   78   4      :   44   Max.   :1.0000   Max.   :1.00000
##  (Other):   77   (Other):   76
##    BILL_AMT3.        BILL_AMT4.       BILL_AMT5.        BILL_AMT6.
##  Min.   :0.00000   Min.   :0.0000   Min.   :0.00000   Min.   :0.0000
##  1st Qu.:0.08805   1st Qu.:0.1969   1st Qu.:0.09219   1st Qu.:0.3281
##  Median :0.09764   Median :0.2159   Median :0.11032   Median :0.3436
##  Mean   :0.11237   Mean   :0.2437   Mean   :0.13494   Mean   :0.3645
##  3rd Qu.:0.11973   3rd Qu.:0.2571   3rd Qu.:0.14598   3rd Qu.:0.3745
##  Max.   :1.00000   Max.   :1.0000   Max.   :1.00000   Max.   :1.0000
##
##    PAY_AMT1.          PAY_AMT2.           PAY_AMT3.
##  Min.   :0.000000   Min.   :0.0000000   Min.   :0.0000000
##  1st Qu.:0.001980   1st Qu.:0.0005937   1st Qu.:0.0009263
##  Median :0.004303   Median :0.0012118   Median :0.0037385
##  Mean   :0.011301   Mean   :0.0035314   Mean   :0.0102255
##  3rd Qu.:0.009956   3rd Qu.:0.0029687   3rd Qu.:0.0090707
##  Max.   :1.000000   Max.   :1.0000000   Max.   :1.0000000
##
##    PAY_AMT4.           PAY_AMT5.           PAY_AMT6.          def_pay
##  Min.   :0.0000000   Min.   :0.0000000   Min.   :0.0000000   0:22357
##  1st Qu.:0.0006164   1st Qu.:0.0007479   1st Qu.:0.0003873   1: 6003
```

```
##  Median :0.0028739   Median :0.0037512   Median :0.0028373
##  Mean   :0.0092567   Mean   :0.0113521   Mean   :0.0099850
##  3rd Qu.:0.0077520   3rd Qu.:0.0098006   3rd Qu.:0.0077317
##  Max.   :1.0000000   Max.   :1.0000000   Max.   :1.0000000
##
```

Splitting the data

```r
library(kknn)
library(class)
set.seed(33)
tr2 = sample(c(1:dim(credit_default2)[1]), 20000)
train2 = credit_default2[tr2,]
test2 = credit_default2[-tr2,]
y_train2=credit_default2[tr2,24]
y_test2=credit_default2[-tr2,24]
```

KNN using out of sample predictions for k=50

```r
set.seed(33)
knn5=knn(train2,test2,cl=y_train2,k=50,prob=FALSE,use.all=FALSE)
knn5_table=table(knn5,y_test2)
accuracy <- function(x){sum(diag(x)/(sum(rowSums(x)))) * 100}

x=accuracy(knn5_table)
precision=posPredValue(knn5,y_test2,positive='1')
recall=sensitivity(knn5,y_test2,positive = '1')
F1=(2*precision*recall)/(precision+recall)

print(knn5_table)
```

```
##     y_test2
## knn5    0    1
##    0 6574  304
##    1   60 1422
```

```r
cat("Accuracy is",x,"and misclassification rate is",100-x)
```

```
## Accuracy is 95.64593 and misclassification rate is 4.354067
```

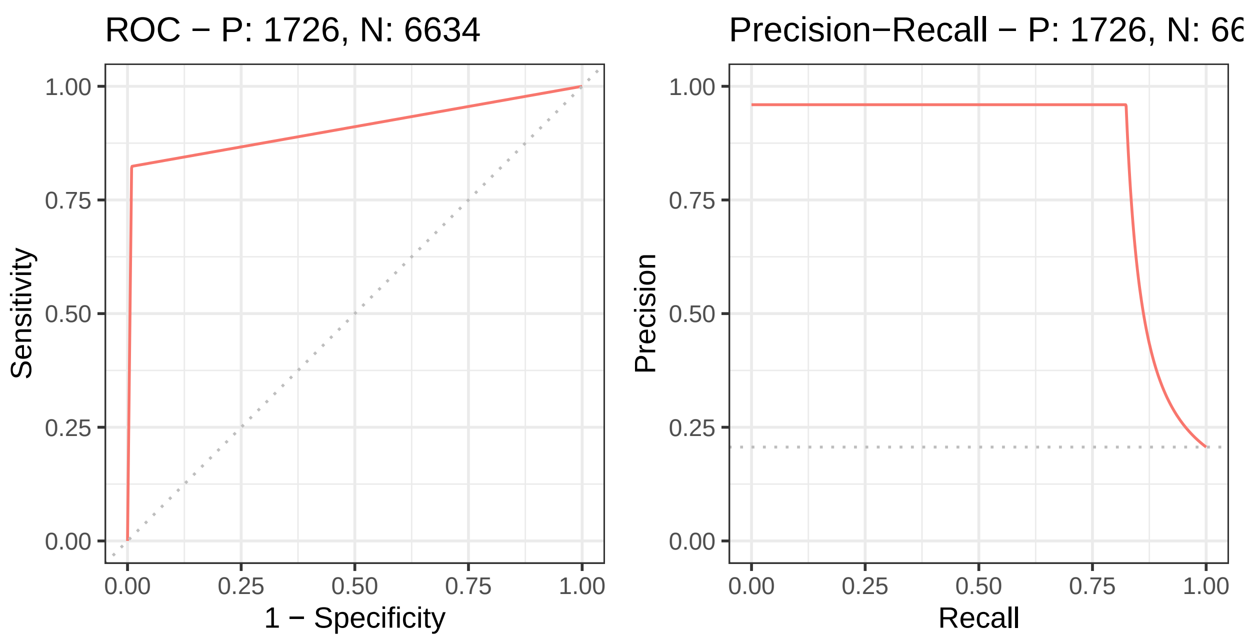
```r
cat("Precision is",precision,"and Recall is",recall)
```

```
## Precision is 0.9595142 and Recall is 0.8238702
```

```r
cat("F1 score is",F1)
```

```
## F1 score is 0.8865337
```

```r
precrec_obj <- evalmod(scores = as.numeric(knn5), labels = y_test2)
autoplot(precrec_obj)
```



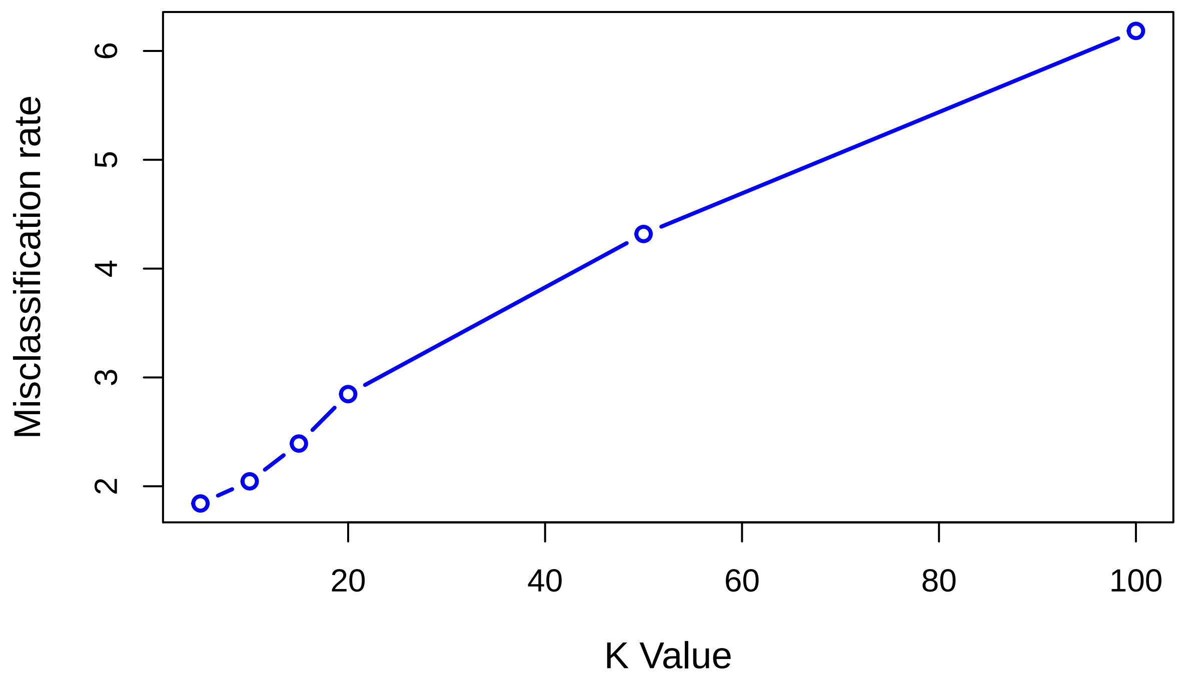
Finding the best k value

```r
set.seed(33)
tr2 = sample(c(1:dim(credit_default2)[1]), 20000)
train2 = credit_default2[tr2,]
test2 = credit_default2[-tr2,]
y_train2=credit_default2[tr2,24]
y_test2=credit_default2[-tr2,24]
k_list2=c(5,10,15,20,50,100)
k_list2
```

```
## [1]   5  10  15  20  50 100
```

```r
y=NULL
for(i in k_list2){
  knn6 = knn(train2,test2,cl=y_train2,k=i)
  knn6_table=table(knn6,y_test2)
  x=accuracy(knn6_table)
  precision=posPredValue(knn6,y_test2,positive='1')
```

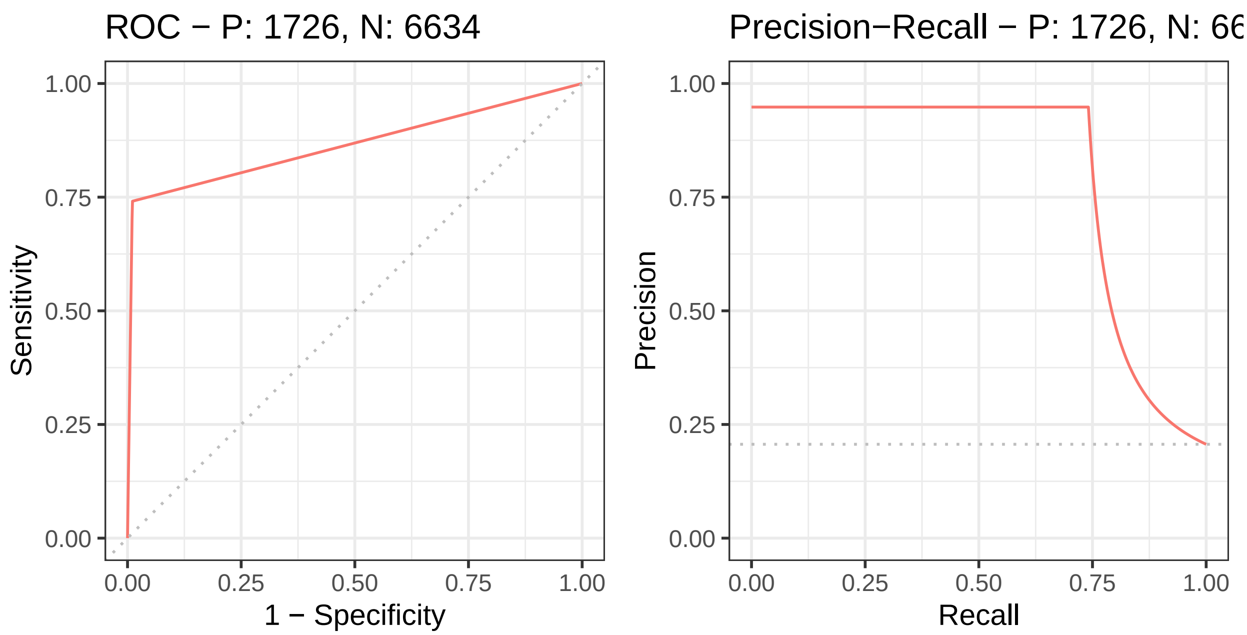
```
  recall=sensitivity(knn6,y_test2,positive = '1')
  F1=(2*precision*recall)/(precision+recall)

  x1=100-x
  y=c(y,x1)
}
best = which.min(y)
plot(k_list2,y,type="b",xlab="K Value",col="blue", ylab="Misclassification rate",lwd=2,cex.lab=1.2)
```




```
cat("Best k value is",k_list2[best],"with misclassification rate of",y[best])
```

```
## Best k value is 5 with misclassification rate of 1.842105
```

```
precrec_obj <- evalmod(scores = as.numeric(knn6), labels = y_test2)
autoplot(precrec_obj)
```



K fold cross validation with kcv=10

```
set.seed(33)
train2 = credit_default2
test2 = credit_default2
y_train2=credit_default2[tr2,24]
y_test2=credit_default2[-tr2,24]
n=dim(credit_default2)[1]
k_list2=c(5,10,20,50,100,150)
kcv = 10
n0 = round(n/kcv,0)
set=1:n
used = NULL
y1=matrix(0,kcv,6)
y2=matrix(0,kcv,6)
y3=matrix(0,kcv,6)
for(j in 1:kcv){
  if(n0<length(set)){val = sample(set,n0)}
  if(n0>=length(set)){val=set}
    train_i = train2[-val,]
    test_i = test2[val,]
    y_train_i=credit_default2[-val,24]
    y_test_i=credit_default2[val,24]
    for(i in 1:6){
```

```r
    knn7 = knn(train_i,test_i,cl=y_train_i,k=k_list2[i])
    knn7_table=table(knn7,y_test_i)

    x1=accuracy(knn7_table)
    x2=100-x1
    precision=posPredValue(knn7,y_test_i,positive='1')
    recall=sensitivity(knn7,y_test_i,positive = '1')
    F1=(2*precision*recall)/(precision+recall)

    cat(x2,"for k value",k_list2[i],"and fold",j,"\n")
    y1[j,i]=x2
    y2[j,i]=precision
    y3[j,i]=recall
   }
  used = union(used,val)
  set = (1:n)[-used]
  cat(j,'\n')
}
```

```
## 1.833568 for k value 5 and fold 1
## 2.115656 for k value 10 and fold 1
## 2.750353 for k value 20 and fold 1
## 4.619182 for k value 50 and fold 1
## 7.157969 for k value 100 and fold 1
## 8.039492 for k value 150 and fold 1
## 1
## 2.186178 for k value 5 and fold 2
## 2.433004 for k value 10 and fold 2
## 2.997179 for k value 20 and fold 2
## 4.54866 for k value 50 and fold 2
## 6.840621 for k value 100 and fold 2
## 8.145275 for k value 150 and fold 2
## 2
## 1.833568 for k value 5 and fold 3
## 2.291961 for k value 10 and fold 3
## 3.244006 for k value 20 and fold 3
## 4.93653 for k value 50 and fold 3
## 6.80536 for k value 100 and fold 3
## 7.651622 for k value 150 and fold 3
## 3
## 1.974612 for k value 5 and fold 4
## 2.468265 for k value 10 and fold 4
## 2.433004 for k value 20 and fold 4
## 4.830748 for k value 50 and fold 4
## 6.205924 for k value 100 and fold 4
## 7.087447 for k value 150 and fold 4
## 4
## 1.798307 for k value 5 and fold 5
## 2.150917 for k value 10 and fold 5
## 2.997179 for k value 20 and fold 5
## 4.866008 for k value 50 and fold 5
## 6.417489 for k value 100 and fold 5
## 7.404795 for k value 150 and fold 5
```

```
## 5
## 1.269394 for k value 5 and fold 6
## 1.480959 for k value 10 and fold 6
## 1.90409 for k value 20 and fold 6
## 3.737659 for k value 50 and fold 6
## 5.183357 for k value 100 and fold 6
## 6.382228 for k value 150 and fold 6
## 6
## 1.868829 for k value 5 and fold 7
## 2.468265 for k value 10 and fold 7
## 2.961918 for k value 20 and fold 7
## 5.077574 for k value 50 and fold 7
## 6.558533 for k value 100 and fold 7
## 8.110014 for k value 150 and fold 7
## 7
## 1.622003 for k value 5 and fold 8
## 1.868829 for k value 10 and fold 8
## 2.433004 for k value 20 and fold 8
## 4.16079 for k value 50 and fold 8
## 5.359661 for k value 100 and fold 8
## 6.699577 for k value 150 and fold 8
## 8
## 2.045134 for k value 5 and fold 9
## 1.868829 for k value 10 and fold 9
## 2.891396 for k value 20 and fold 9
## 4.196051 for k value 50 and fold 9
## 6.170663 for k value 100 and fold 9
## 7.475317 for k value 150 and fold 9
## 9
## 1.974612 for k value 5 and fold 10
## 2.503526 for k value 10 and fold 10
## 3.455571 for k value 20 and fold 10
## 4.971791 for k value 50 and fold 10
## 6.311707 for k value 100 and fold 10
## 7.651622 for k value 150 and fold 10
## 10
```

```r
my1=apply(y1,2,mean)
my2=apply(y2,2,mean)
my3=apply(y3,2,mean)
cat("Misclassification rate values:",my1)
```

```
## Misclassification rate values: 1.840621 2.165021 2.80677 4.594499 6.301128 7.464739
```

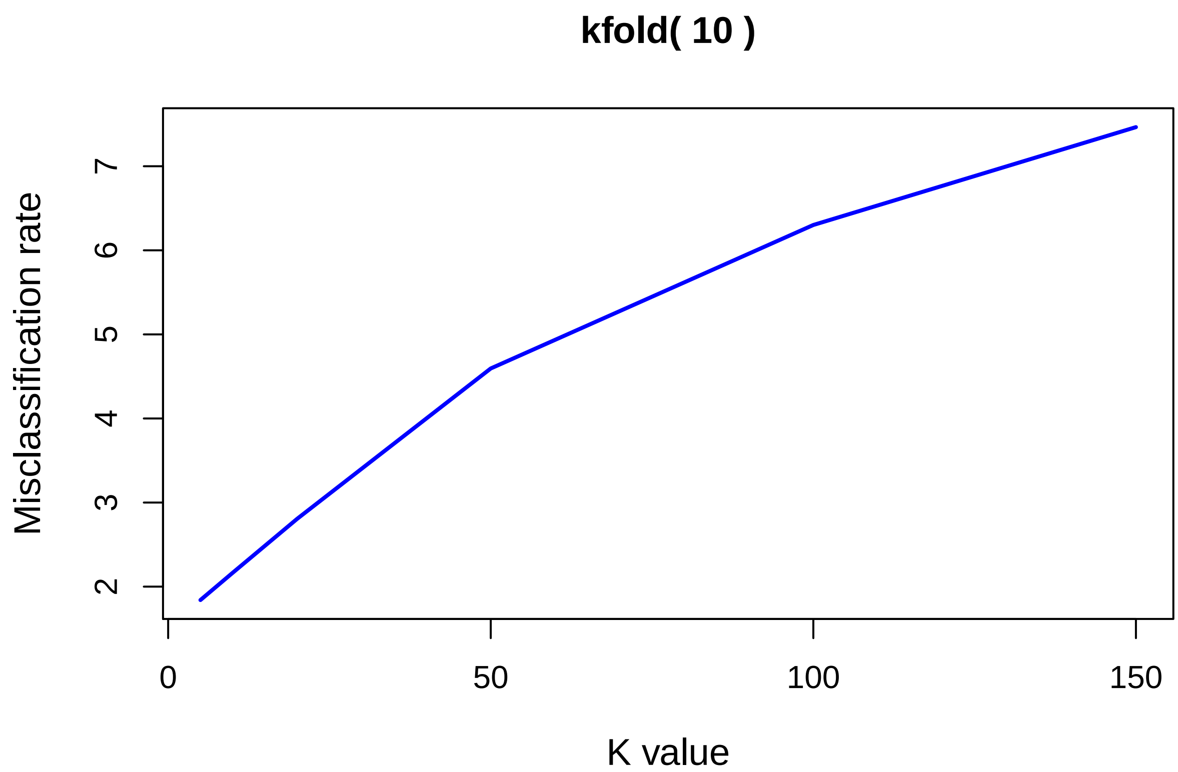
```r
cat("Precision values:",my2)
```

```
## Precision values: 0.9800631 0.9751438 0.9696035 0.9602635 0.952159 0.9437626
```

```r
cat("Recall values:",my3)
```

```
## Recall values: 0.9321262 0.9213272 0.8958646 0.8168273 0.7398256 0.6888161
```

```r
best = which.min(my1)
plot(k_list2,my1,xlab="K value",ylab="Misclassification rate",col=4,lwd=2,type="l",cex.lab=1.2,main=pas
```




```r
cat("Best k value is",k_list2[best],"with misclassification rate of",my1[best])
```

```
## Best k value is 5 with misclassification rate of 1.840621
```

K fold cross validation with kcv=5

```r
set.seed(33)
train2 = credit_default2
test2 = credit_default2
y_train2=credit_default2[tr2,24]
y_test2=credit_default2[-tr2,24]
n=dim(credit_default2)[1]
k_list2=c(5,10,20,50,100,150)
kcv = 5
n0 = round(n/kcv,0)
set=1:n
used = NULL
y1=matrix(0,kcv,6)
y2=matrix(0,kcv,6)
```

```
y3=matrix(0,kcv,6)
for(j in 1:kcv){
  if(n0<length(set)){val = sample(set,n0)}
  if(n0>=length(set)){val=set}
    train_i = train2[-val,]
    test_i = test2[val,]
    y_train_i=credit_default2[-val,24]
    y_test_i=credit_default2[val,24]
    for(i in 1:6){
     knn7 = knn(train_i,test_i,cl=y_train_i,k=k_list2[i])
     knn7_table=table(knn7,y_test_i)

     x1=accuracy(knn7_table)
     x2=100-x1
     precision=posPredValue(knn7,y_test_i,positive='1')
     recall=sensitivity(knn7,y_test_i,positive = '1')
     F1=(2*precision*recall)/(precision+recall)

     cat(x2,"for k value",k_list2[i],"and fold",j,"\n")
     y1[j,i]=x2
     y2[j,i]=precision
     y3[j,i]=recall
    }
  used = union(used,val)
  set = (1:n)[-used]
  cat(j,'\n')
}
```

```
## 2.027504 for k value 5 and fold 1
## 2.291961 for k value 10 and fold 1
## 3.42031 for k value 20 and fold 1
## 5.200987 for k value 50 and fold 1
## 6.999295 for k value 100 and fold 1
## 7.810296 for k value 150 and fold 1
## 1
## 1.851199 for k value 5 and fold 2
## 2.309591 for k value 10 and fold 2
## 3.102962 for k value 20 and fold 2
## 4.989422 for k value 50 and fold 2
## 5.976728 for k value 100 and fold 2
## 7.21086 for k value 150 and fold 2
## 2
## 1.745416 for k value 5 and fold 3
## 2.009873 for k value 10 and fold 3
## 2.591678 for k value 20 and fold 3
## 4.583921 for k value 50 and fold 3
## 6.170663 for k value 100 and fold 3
## 7.475317 for k value 150 and fold 3
## 3
## 1.851199 for k value 5 and fold 4
## 2.291961 for k value 10 and fold 4
## 2.856135 for k value 20 and fold 4
## 4.707334 for k value 50 and fold 4
```

```
## 6.311707 for k value 100 and fold 4
## 7.810296 for k value 150 and fold 4
## 4
## 1.939351 for k value 5 and fold 5
## 2.485896 for k value 10 and fold 5
## 3.490832 for k value 20 and fold 5
## 5.430183 for k value 50 and fold 5
## 7.246121 for k value 100 and fold 5
## 8.515515 for k value 150 and fold 5
## 5
```

```r
my1=apply(y1,2,mean)
my2=apply(y2,2,mean)
my3=apply(y3,2,mean)
cat("Misclassification rate values:",my1)
```

```
## Misclassification rate values: 1.882934 2.277856 3.092384 4.98237 6.540903 7.764457
```

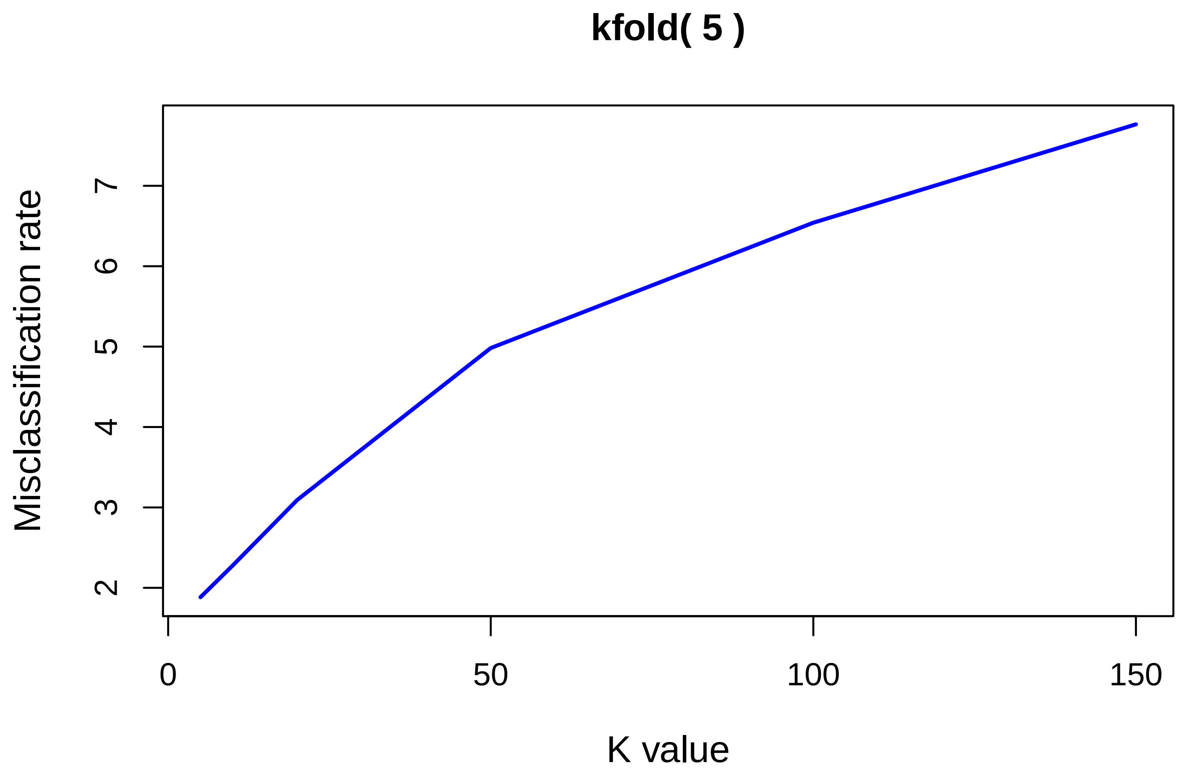
```r
cat("Precision values:",my2)
```

```
## Precision values: 0.9796457 0.9746248 0.968322 0.9578953 0.9527022 0.9397639
```

```r
cat("Recall values:",my3)
```

```
## Recall values: 0.9303466 0.9162723 0.8828379 0.7997678 0.7273391 0.6768292
```

```r
best = which.min(my1)
plot(k_list2,my1,xlab="K value",ylab="Misclassification rate",col=4,lwd=2,type="l",cex.lab=1.2,main=past
```




```
cat("Best k value is",k_list2[best],"with misclassification rate of",my1[best])
```

```
## Best k value is 5 with misclassification rate of 1.882934
```

KNN with normalization and SMOTE

Reading the data

```
rm(list=ls())

credit_default3=read.csv('cleansed_data.csv',header=T)
credit_default3=credit_default3[-1]
```

Converting categorical variables

```
factor_vars3 = c(2, 3, 4, 24, c(6:11))
for (i in factor_vars3) {
credit_default3[[i]]<-as.factor(credit_default3[[i]])
}
```

Normalization

```
normalize <- function(x) {
return ((x - min(x)) / (max(x) - min(x))) }
normalize_vars=c(1,5,c(12:23))
for (i in normalize_vars) {
credit_default3[[i]]=normalize(credit_default3[[i]])
}
```

Splitting the data

```
library(kknn)
library(class)
set.seed(33)
tr3 = sample(c(1:dim(credit_default3)[1]), 20000)
train3 = credit_default3[tr3,]
test3 = credit_default3[-tr3,]
y_train3=credit_default3[tr3,24]
y_test3=credit_default3[-tr3,24]
prop.table(table(y_test3))
```

```
## y_test3
##         0         1
## 0.7935407 0.2064593
```

```
library(DMwR)
```

```
## Loading required package: grid
```

```
## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo
```

```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:data.table':
##
##     between, first, last
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
train3$def_pay <- as.factor(train3$def_pay)
train3 <- SMOTE(train3$def_pay ~ ., train3, perc.over = 100, perc.under=200)
train3$def_pay <- as.numeric(train3$def_pay)
train3$def_pay=ifelse(train3$def_pay==2,1,0)
prop.table(table(train3$def_pay))
```

```
##
##   0   1
## 0.5 0.5
```

```
library(caret)
library(precrec)
library(ROCit)
library(plotROC)
library(ggplot2)

set.seed(33)
knn9=knn(train3,test3,cl=train3$def_pay,k=50,prob=FALSE,use.all=FALSE)
knn9_table=table(knn9,y_test3)
accuracy <- function(x){sum(diag(x)/(sum(rowSums(x)))) * 100}

x=accuracy(knn9_table)
precision=posPredValue(knn9,test3$def_pay,positive='1')
recall=sensitivity(knn9,test3$def_pay,positive = '1')
F1=(2*precision*recall)/(precision+recall)


print(knn9_table)
```

```
##     y_test3
## knn9    0    1
##    0 6130   46
##    1  504 1680
```

```
cat("Accuracy is",x,"and misclassification rate is",100-x)
```

```
## Accuracy is 93.42105 and misclassification rate is 6.578947
```

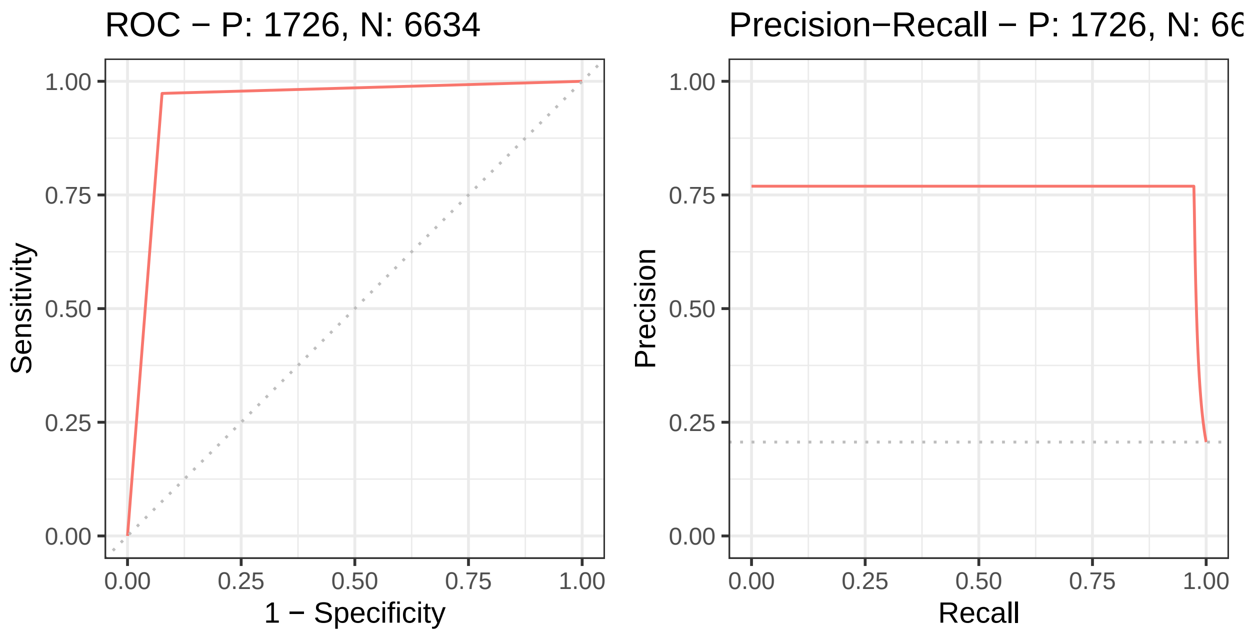
```
cat("Precision is",precision,"and Recall is",recall)
```

```
## Precision is 0.7692308 and Recall is 0.9733488
```

```
cat("F1 score is",F1)
```

```
## F1 score is 0.859335
```

```
precrec_obj <- evalmod(scores = as.numeric(knn9), labels = y_test3)
autoplot(precrec_obj)
```



K fold cross validation with kcv=10

```
set.seed(33)
test3 = credit_default3
y_test3=credit_default3[-tr3,24]
n=dim(credit_default3)[1]
k_list3=c(5,10,20,50,100,150)
kcv = 10
n0 = round(n/kcv,0)
set=1:n
used = NULL
y1=matrix(0,kcv,6)
y2=matrix(0,kcv,6)
y3=matrix(0,kcv,6)
for(j in 1:kcv){
  if(n0<length(set)){val = sample(set,n0)}
  if(n0>=length(set)){val=set}
    train_i = train3[-val,]
    test_i = test3[val,]
    y_test_i=credit_default3[val,24]
```

```r
    for(i in 1:6){
     knn10 = knn(train_i,test_i,cl=train_i$def_pay,k=k_list3[i])
     knn10_table=table(knn10,y_test_i)

     x1=accuracy(knn10_table)
     x2=100-x1
     precision=posPredValue(knn10,y_test_i,positive='1')
     recall=sensitivity(knn10,y_test_i,positive = '1')
     F1=(2*precision*recall)/(precision+recall)

     cat(x2,"for k value",k_list3[i],"and fold",j,"\n")
     y1[j,i]=x2
     y2[j,i]=precision
     y3[j,i]=recall
    }
  used = union(used,val)
  set = (1:n)[-used]
  cat(j,'\n')
}
```

```
## 2.856135 for k value 5 and fold 1
## 3.244006 for k value 10 and fold 1
## 4.654443 for k value 20 and fold 1
## 7.157969 for k value 50 and fold 1
## 8.815233 for k value 100 and fold 1
## 9.767278 for k value 150 and fold 1
## 1
## 2.679831 for k value 5 and fold 2
## 3.067701 for k value 10 and fold 2
## 4.019746 for k value 20 and fold 2
## 6.664316 for k value 50 and fold 2
## 7.96897 for k value 100 and fold 2
## 8.850494 for k value 150 and fold 2
## 2
## 2.433004 for k value 5 and fold 3
## 2.715092 for k value 10 and fold 3
## 4.055007 for k value 20 and fold 3
## 6.699577 for k value 50 and fold 3
## 8.251058 for k value 100 and fold 3
## 9.132581 for k value 150 and fold 3
## 3
## 2.362482 for k value 5 and fold 4
## 3.208745 for k value 10 and fold 4
## 4.54866 for k value 20 and fold 4
## 7.263752 for k value 50 and fold 4
## 9.09732 for k value 100 and fold 4
## 9.414669 for k value 150 and fold 4
## 4
## 2.64457 for k value 5 and fold 5
## 3.279267 for k value 10 and fold 5
## 4.337094 for k value 20 and fold 5
## 7.757405 for k value 50 and fold 5
## 9.308886 for k value 100 and fold 5
```

```
## 9.908322 for k value 150 and fold 5
## 5
## 2.609309 for k value 5 and fold 6
## 2.538787 for k value 10 and fold 6
## 3.878702 for k value 20 and fold 6
## 6.135402 for k value 50 and fold 6
## 8.074753 for k value 100 and fold 6
## 8.462623 for k value 150 and fold 6
## 6
## 2.891396 for k value 5 and fold 7
## 3.455571 for k value 10 and fold 7
## 5.112835 for k value 20 and fold 7
## 7.052186 for k value 50 and fold 7
## 8.638928 for k value 100 and fold 7
## 9.48519 for k value 150 and fold 7
## 7
## 2.221439 for k value 5 and fold 8
## 2.891396 for k value 10 and fold 8
## 4.301834 for k value 20 and fold 8
## 6.558533 for k value 50 and fold 8
## 8.110014 for k value 100 and fold 8
## 8.674189 for k value 150 and fold 8
## 8
## 2.433004 for k value 5 and fold 9
## 2.574048 for k value 10 and fold 9
## 3.667137 for k value 20 and fold 9
## 6.100141 for k value 50 and fold 9
## 7.475317 for k value 100 and fold 9
## 7.863188 for k value 150 and fold 9
## 9
## 2.327221 for k value 5 and fold 10
## 2.926657 for k value 10 and fold 10
## 3.878702 for k value 20 and fold 10
## 6.80536 for k value 50 and fold 10
## 8.32158 for k value 100 and fold 10
## 9.344147 for k value 150 and fold 10
## 10
```

```r
my1=apply(y1,2,mean)
my2=apply(y2,2,mean)
my3=apply(y3,2,mean)
cat("Misclassification rate values:",my1)
```

```
## Misclassification rate values: 2.545839 2.990127 4.245416 6.819464 8.406206 9.090268
```

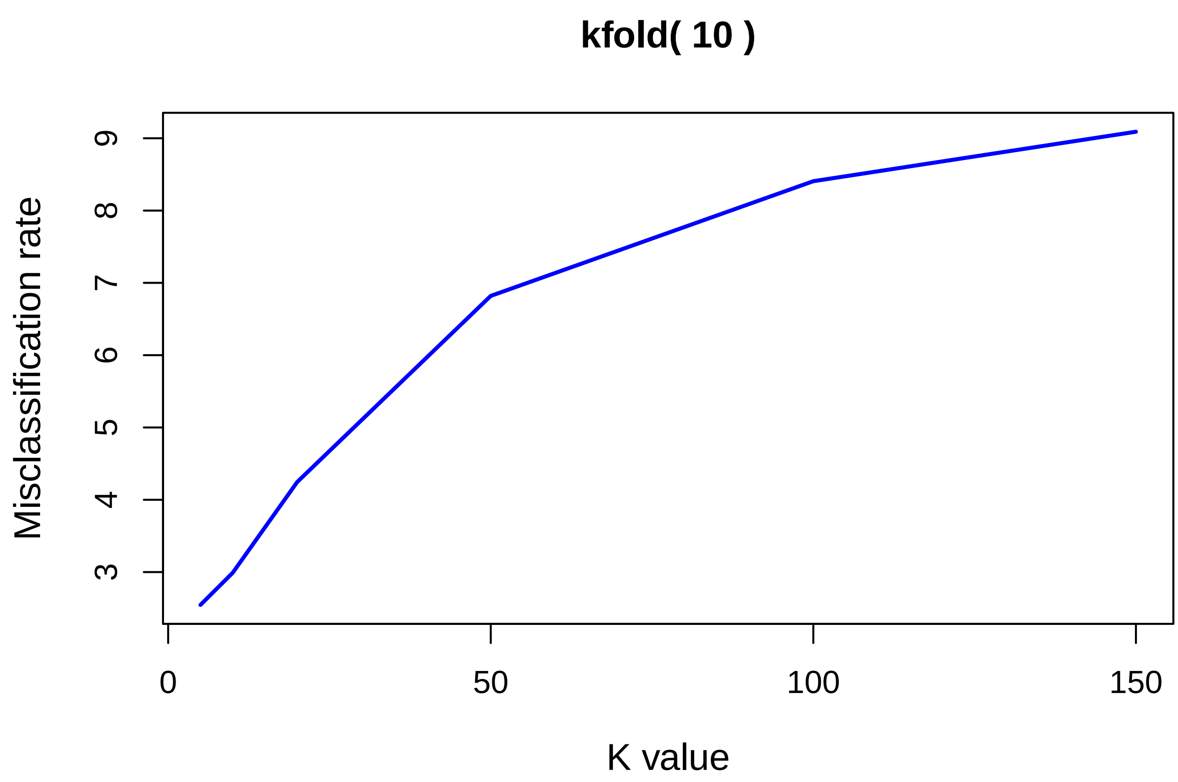
```r
cat("Precision values:",my2)
```

```
## Precision values: 0.9033111 0.8856398 0.8409694 0.7683876 0.7354873 0.7270599
```

```r
cat("Recall values:",my3)
```

```
## Recall values: 0.9849576 0.985847 0.9853982 0.9702046 0.9414402 0.9132637
```

```
best = which.min(my1)
plot(k_list3,my1,xlab="K value",ylab="Misclassification rate",col=4,lwd=2,type="l",cex.lab=1.2,main=pas
```




```
cat("Best k value is",k_list3[best],"with misclassification rate of",my1[best])
```

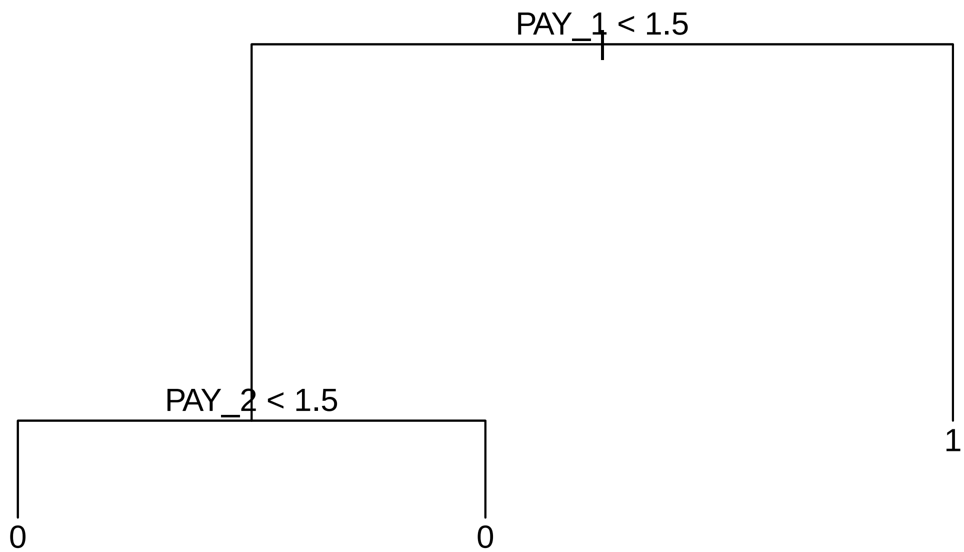
```
## Best k value is 5 with misclassification rate of 2.545839
```

#PART - 3 DECISION TREES

```
# Fitting Classification Trees
rm(list=ls())
library(tree)
df = read.csv('cleansed_data.csv')
df$ID = NULL

train_sample = sample(c(1:dim(df)[1]), 20000)
train = df[train_sample,]
test = df[-train_sample,]
ytrain = train$def_pay
ytest = test$def_pay

tree.credit = tree(factor(def_pay)~.,data=train)
summary(tree.credit)
```

```
##
## Classification tree:
## tree(formula = factor(def_pay) ~ ., data = train)
## Variables actually used in tree construction:
## [1] "PAY_1" "PAY_2"
## Number of terminal nodes:  3
## Residual mean deviance:  0.8564 = 17130 / 20000
## Misclassification error rate: 0.1674 = 3348 / 20000
```

```
plot(tree.credit)
text(tree.credit, pretty =0)
```

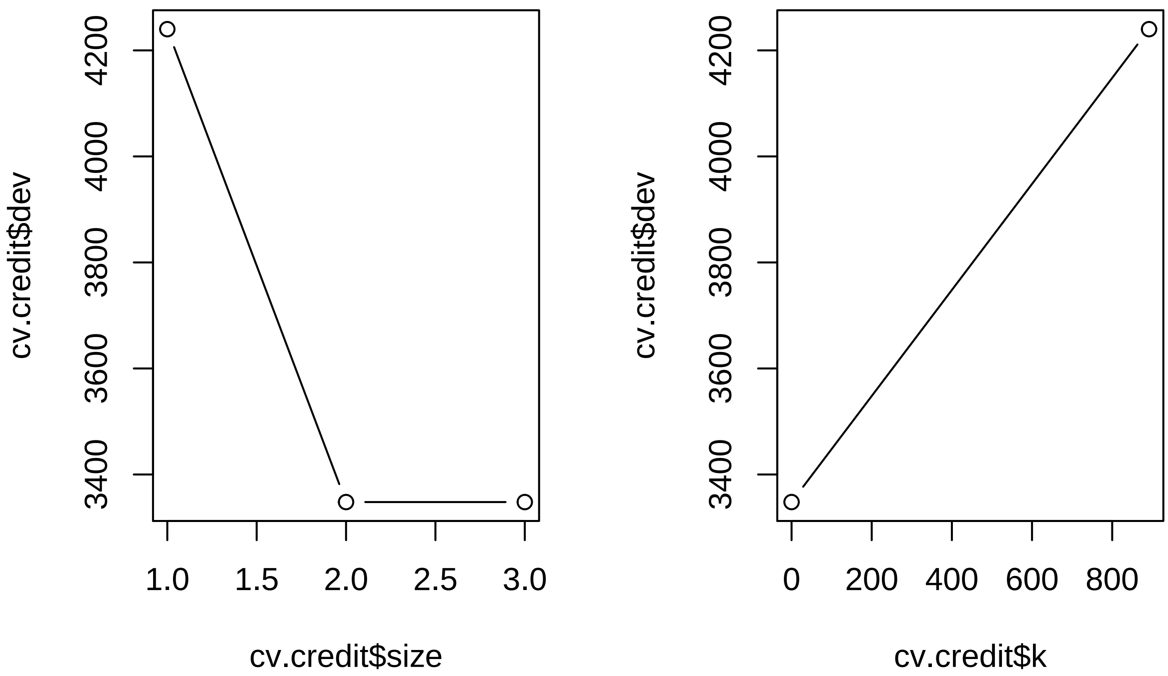



```
#In order to properly evaluate the performance of a classification tree on these data
#We must estimate the test error rather than simply computing the training error.
tree.pred = predict(tree.credit, test ,type ="class")
table(tree.pred, ytest)
```

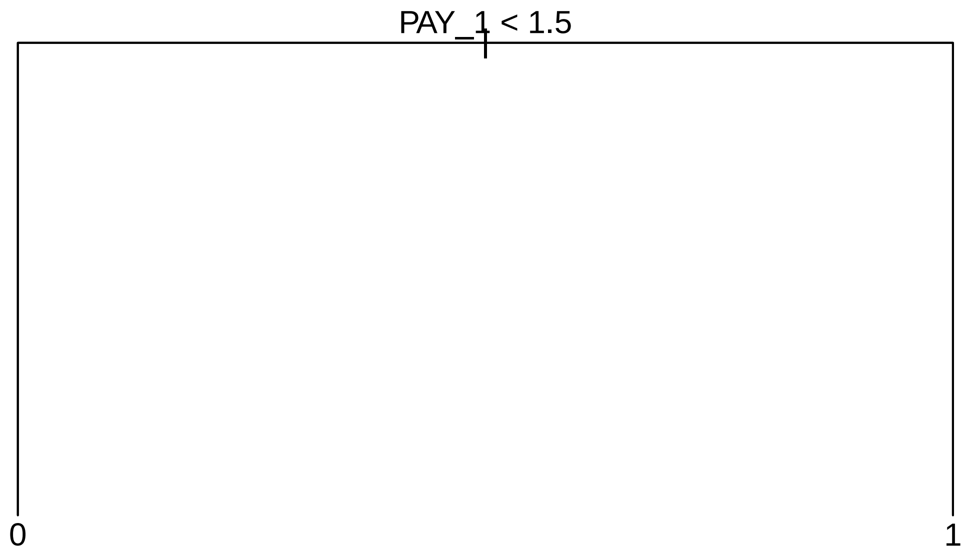
```
##          ytest
## tree.pred    0    1
##         0 6315 1156
##         1  282  607
```

```
#The function cv.tree() performs cross-validation in order to
#cv.tree() determine the optimal level of tree complexity;
```

```r
set.seed (3)
cv.credit =cv.tree(tree.credit,FUN=prune.misclass)

par(mfrow =c(1,2))
plot(cv.credit$size, cv.credit$dev ,type="b")
plot(cv.credit$k, cv.credit$dev ,type="b")
```




```r
prune.credit =prune.misclass (tree.credit, best =2)
plot(prune.credit)
text(prune.credit,pretty =0)
```




```r
summary(prune.credit)
```

```
##
## Classification tree:
## snip.tree(tree = tree.credit, nodes = 2L)
## Variables actually used in tree construction:
## [1] "PAY_1"
## Number of terminal nodes:  2
## Residual mean deviance:  0.8926 = 17850 / 20000
## Misclassification error rate: 0.1674 = 3348 / 20000
```

```r
tree.pred.prune=predict (prune.credit, test ,type="class")
table(tree.pred.prune, ytest)
```

```
##                ytest
## tree.pred.prune    0    1
##               0 6315 1156
##               1  282  607
```

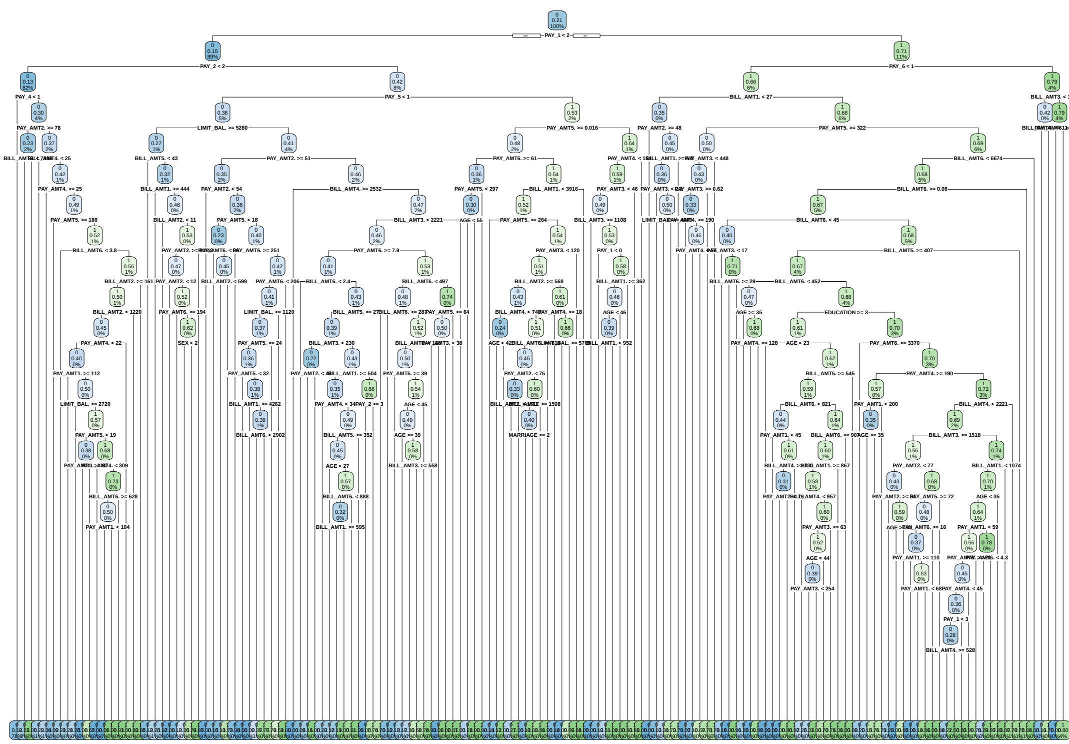
```r
################################################################################
library(tree)
library(rpart)
library(rpart.plot)
```

```r
#fit a big tree using rpart.control
big.tree = rpart(def_pay ~ .,method="class",data=train,
                 control=rpart.control(minsplit=5,cp=.0005))
nbig = length(unique(big.tree$where))
cat('size of big tree: ',nbig,'\n')
```

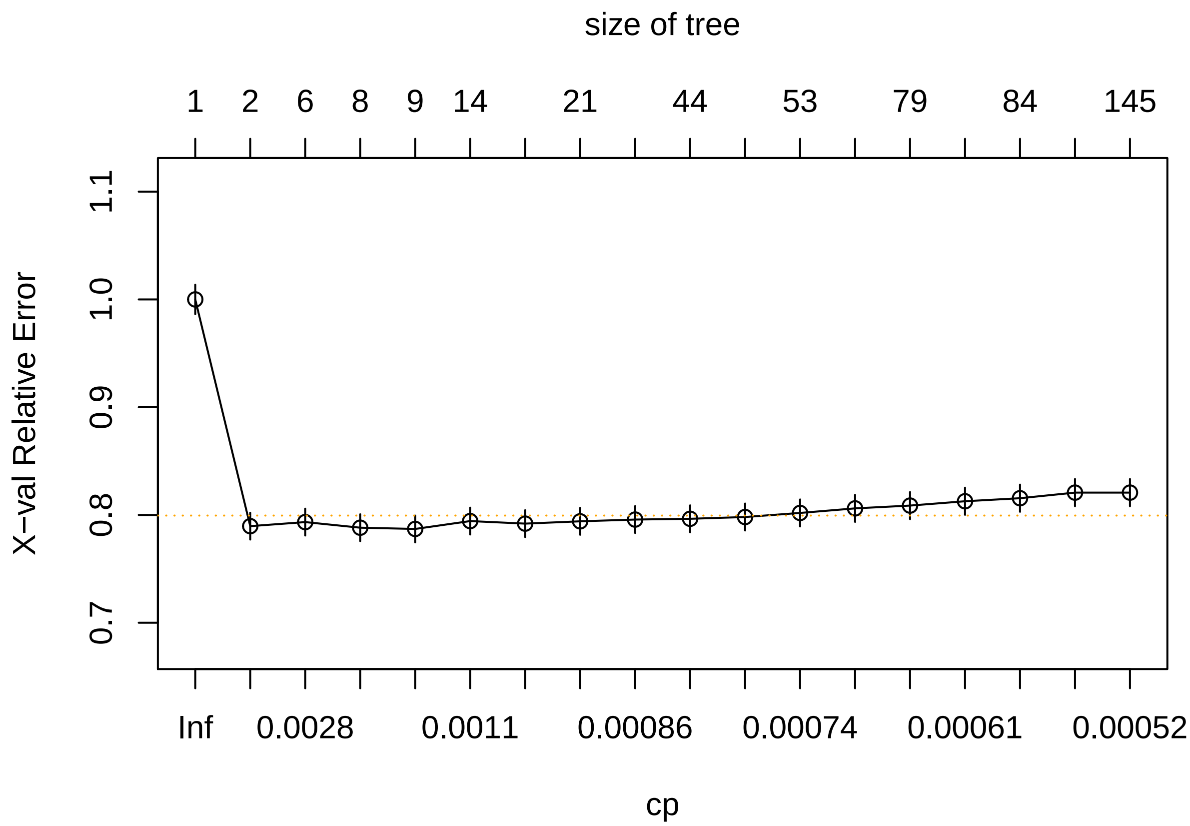
```
## size of big tree:  145
```

```r
rpart.plot(big.tree)
```

```
## Warning: labs do not fit even at cex 0.15, there may be some overplotting
```



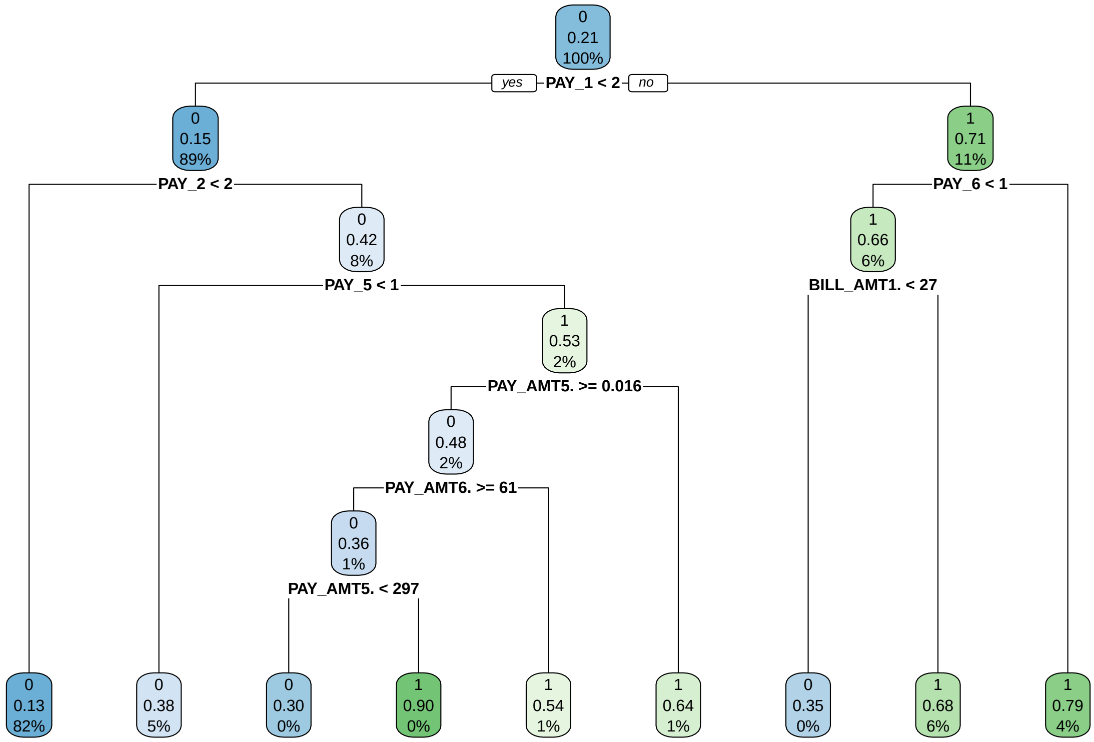
```r
#-----------------------------------------------------
#look at cross-validation
par(mfrow=c(1,1))
plotcp(big.tree, col = 'orange')
```




```
#--------------------------------------------------
#show fit from some trees
bestcp=big.tree$cptable[which.min(big.tree$cptable[,"xerror"]),"CP"]
cat('bestcp: ',bestcp,'\n')
```

```
## bestcp:  0.001120283
```

```
ptree = prune(big.tree,cp=bestcp)
rpart.plot(ptree)
```



PART 4 - RANDOM FOREST

```r
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

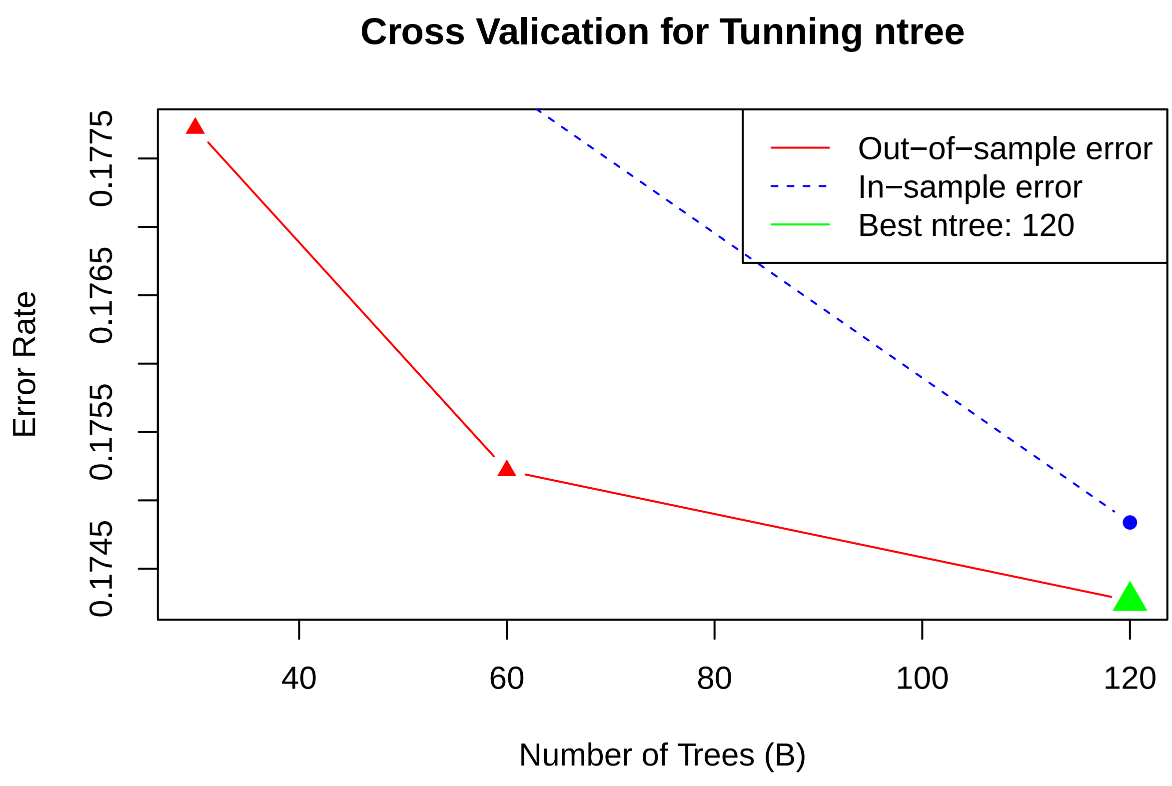
```r
# Read in data
data = read.csv("cleansed_data.csv")

# Take Id out of dataset
credit.card = data[, !names(data) == "ID"]

# Factorize the categorical variables
cate_vari = c(2:4,6:11,24)
for (i in cate_vari){
  credit.card[[i]] = as.factor(credit.card[[i]])
}

# Split the data into training set and testing set
# We agree on using 2/3 data for train
train = sample(1:nrow(credit.card), 20000)
test = (-train)
```

```r
# Pick n_tree using cross validation.
# We tried: n.list = seq(50,1000,length=20) | k=6 for extual result which shows in graph and yieds best
n.list = c(30, 60, 120)
k = 3
n.samp = round(nrow(credit.card)/k) # determine the train size for k-fold
iAccuracyMx = matrix(0, k, length(n.list)) # matrix records in sample accuracy
accuracyMx = matrix(0, k, length(n.list)) # out-of-sample accuracy matrix
used = NULL
set = 1:nrow(credit.card)
for (ki in 1:k){
  # determine number of each fold in case data cant be split evenly for each fold
  if (length(set)>=n.samp) {val=n.samp}
  if (length(set)<n.samp) {val=length(set)}
  test.cv = sample(set, val)
  train.cv = (-test.cv)
  for (ni in 1:length(n.list)){
    cat('Now is processing:', ki, ni, "\n")
    set.seed(7)
    rf.fit = randomForest(def_pay~., data = credit.card[train.cv,],
                          ntree = n.list[ni], importance = T, mtry=23)
    rf.pred = predict(rf.fit, newdata = credit.card[test.cv,])
    # Geting in-sample accuracy
    iAccuracyMx[ki, ni] = mean(rf.fit$predicted==credit.card$def_pay[train.cv])
    # Getting out-of sample accuracy
    accuracyMx[ki, ni] = mean(rf.pred==credit.card$def_pay[test.cv])

  used = union(used, test.cv) # update valication set for other folds
  set = (1:nrow(credit.card))[-used]
  }
}
```

```
## Now is processing: 1 1
## Now is processing: 1 2
## Now is processing: 1 3
## Now is processing: 2 1
## Now is processing: 2 2
```

```
## Now is processing: 2 3
## Now is processing: 3 1
## Now is processing: 3 2
## Now is processing: 3 3
```

```r
# Calculate the average error for each ntree
mError = 1-apply(accuracyMx, 2, mean) # Geting out-of-sample misclassification
imError = 1-apply(iAccuracyMx, 2, mean) # Geting in sample misclassification
intree = which.min(mError) # index for best ntree

# Plot the both in sample and out of sample error rates
plot(n.list, mError, col="red", pch=17, type = 'b',
     main="Cross Valication for Tunning ntree",
     xlab = "Number of Trees (B)",
     ylab = "Error Rate")
lines(n.list, imError, col="blue", pch=16, type = 'b', lty=2)
points(n.list[intree], min(mError), col = "green", pch = 17, cex = 1.8)
legend("topright",
       c("Out-of-sample error", "In-sample error", "Best ntree: 120"),
       col = c("red", "blue", "green"), lty = 1:2)
```

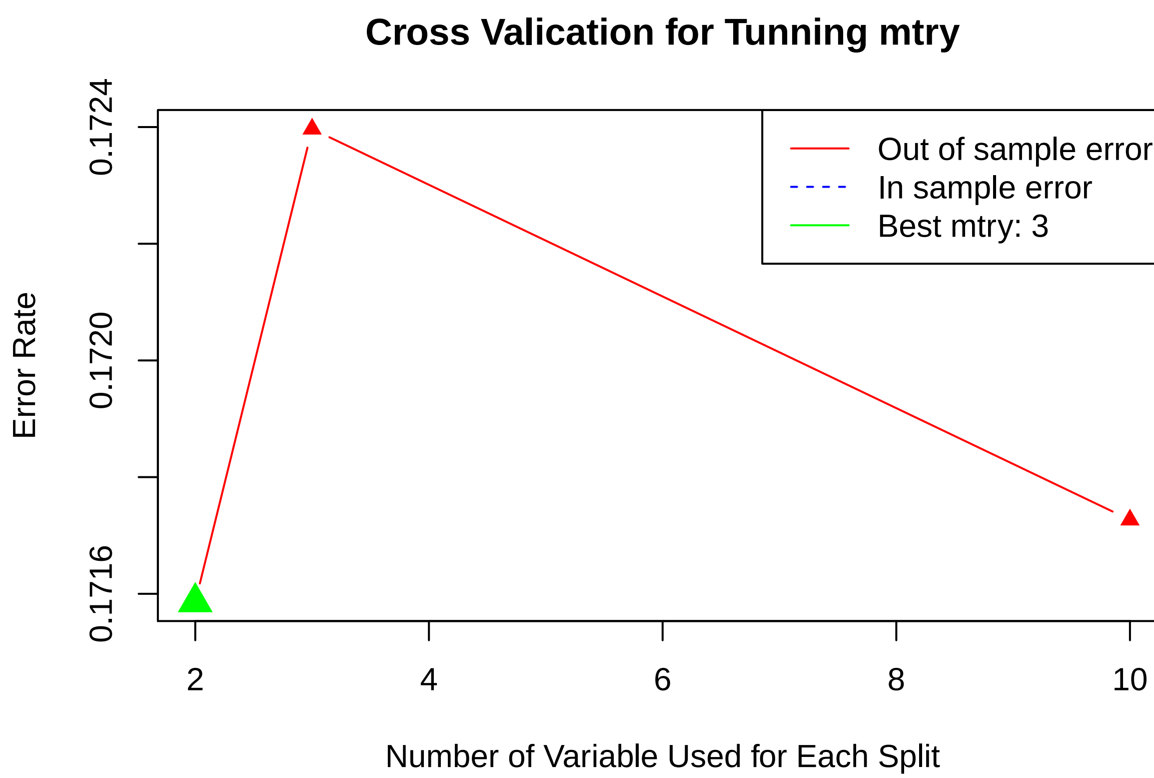



```r
# Selecting best mtry using cross validation
mtry.list = c(2,3,10) # tried mtry.list = 1:23 | k =5 for actual analysis & got best mtry = 3. Here usi
k2 = 3
n.samp2 = round(nrow(credit.card)/k2)
```

```r
iAccuracyMx2 = matrix(0, k2, length(mtry.list))
accuracyMx2 = matrix(0, k2, length(mtry.list))
used = NULL
set = 1:nrow(credit.card)
for (ki in 1:k2){
  if (length(set)>=n.samp2) {val=n.samp2}
  if (length(set)<n.samp2) {val=length(set)}
  test.cv = sample(set, val)
  train.cv = (-test.cv)
  for (mti in 1:length(mtry.list)){
    cat('Now is processing:', ki, mti, "\n")
    set.seed(7)
    rf.fit = randomForest(def_pay~., data = credit.card[train.cv,],
                          ntree = 120, importance = T,
                          mtry = mtry.list[mti]) # in actual analysis use ntree=600
    rf.pred = predict(rf.fit, newdata = credit.card[test.cv,])
    iAccuracyMx2[ki, mti]= mean(rf.fit$predicted==credit.card$def_pay[train.cv])
    accuracyMx2[ki, mti] = mean(rf.pred==credit.card$def_pay[test.cv])
  used = union(used, test.cv)
  set = (1:nrow(credit.card))[-used]
  }
}
```

```
## Now is processing: 1 1
## Now is processing: 1 2
## Now is processing: 1 3
## Now is processing: 2 1
## Now is processing: 2 2
## Now is processing: 2 3
## Now is processing: 3 1
## Now is processing: 3 2
## Now is processing: 3 3
```

```r
# Calculate the average error for each mtry
mError2 = 1-apply(accuracyMx2, 2, mean)
imError2 = 1-apply(iAccuracyMx2, 2, mean)
imtry = which.min(mError2) # index of best mtry

# Plot the both in sample and out of sample errors
plot(mtry.list, mError2, col="red", pch=17, type = 'b',
     main="Cross Valication for Tunning mtry",
     xlab = "Number of Variable Used for Each Split",
     ylab = "Error Rate")
lines(mtry.list, imError2, col="blue", pch=16, type = 'b', lty=2)
points(mtry.list[imtry], min(mError2), col = "green", pch = 17, cex = 1.8)
legend("topright",
       c("Out of sample error", "In sample error", "Best mtry: 3"),
       col = c("red", "blue", "green"), lty = 1:2)
```




```r
# Use the best mtry = 3, and best ntree = 600 retrain the data
rf.fit_best = randomForest(def_pay~., data = credit.card[train,],mtry=3,ntree = 120, importance = T) #
# Predict test set
set.seed(7)
rf.pred_best = predict(rf.fit_best, newdata = credit.card[test,])
table(rf.pred_best, credit.card$def_pay[test]) # make confusion matrix
```

```
##
## rf.pred_best    0    1
##            0 6243 1046
##            1  379  692
```

```r
cat("Random forest correct rate with mtry = 3:", # Showing accuracy
    mean(rf.pred_best ==credit.card$def_pay[test]))
```

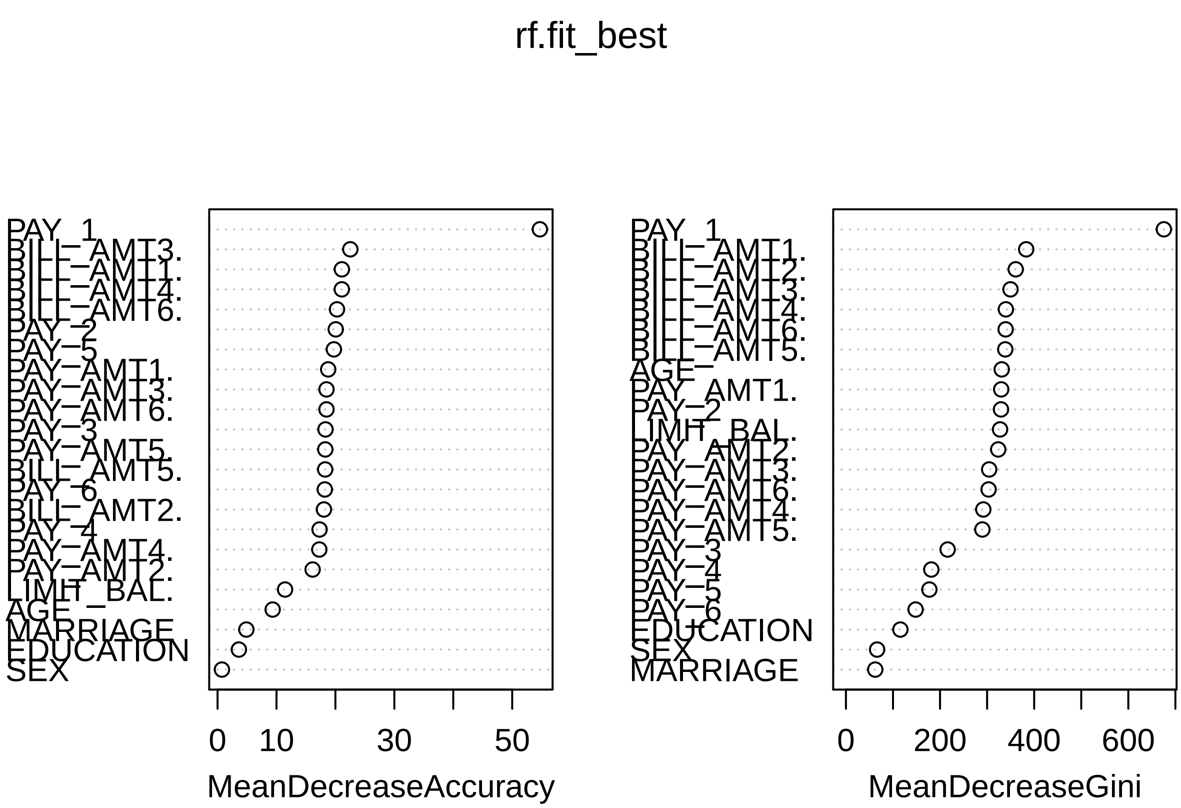
```
## Random forest correct rate with mtry = 3: 0.8295455
```

```r
# Check importance
importance(rf.fit_best)
```

```
##                     0          1 MeanDecreaseAccuracy MeanDecreaseGini
## LIMIT_BAL.  7.2796755 10.4892631           11.4602993        327.53869
## SEX        -0.2191581  1.9231012            0.7421013         66.29946
## EDUCATION   3.6162822  1.0787533            3.6188925        115.69248
```

```
## MARRIAGE    5.4094753 -0.2396775            4.8886092         62.28436
## AGE        11.4688384 -1.3503555            9.3511068        331.07555
## PAY_1      55.3694916 12.7850491           54.6970554        675.60402
## PAY_2      19.0565054  4.6829475           20.0576719        329.53380
## PAY_3      17.9610933  2.5044267           18.3167375        216.14728
## PAY_4      18.7223832  1.5621189           17.3092592        181.42442
## PAY_5      18.0616702  4.0989853           19.7396425        177.32760
## PAY_6      17.1857401  4.9835119           18.2033100        148.06632
## BILL_AMT1. 17.2367735  6.2976864           21.0941108        383.06125
## BILL_AMT2. 16.2541507 -0.6652245           18.0504349        360.79963
## BILL_AMT3. 20.4709200 -1.3034772           22.5286782        349.67504
## BILL_AMT4. 17.0578996  2.9010147           21.0834602        339.79458
## BILL_AMT5. 15.5152829  0.7880880           18.2662038        338.59787
## BILL_AMT6. 17.7250913  2.7741325           20.2709440        339.34640
## PAY_AMT1.  16.8606768 -1.9608058           18.7738616        329.95730
## PAY_AMT2.  14.2277268  2.0774538           16.1330486        323.59554
## PAY_AMT3.  16.0854870  1.6645380           18.4862391        304.42943
## PAY_AMT4.  16.3526538  1.7873737           17.2705467        291.89817
## PAY_AMT5.  16.4428282  1.5651727           18.2821824        289.95395
## PAY_AMT6.  18.6945017  1.6050130           18.4855416        303.13594
```

```
varImpPlot(rf.fit_best)
```



#PART 5 - XG BOOST

process data

```r
library(xgboost)
```

```
##
## Attaching package: 'xgboost'

## The following object is masked from 'package:dplyr':
##
##     slice
```

```r
df = read.csv('cleansed_data.csv')
df$ID = NULL


# factorize
# factor_vars1 = c(2, 3, 4, 24, c(6:11))
# for (i in factor_vars1) {
# df[[i]]<-as.factor(df[[i]])
# }

## sampling
set.seed(33)
tr = sample(c(1:dim(df)[1]), 20000)
train = df[tr,]
test = df[-tr,]
y_tr = train$def_pay
y_ts = test$def_pay

dim(df)
```

```
## [1] 28360    24
```

```r
summary(df)
```

```
##    LIMIT_BAL.          SEX          EDUCATION      MARRIAGE
##  Min.   :  320   Min.   :1.000   Min.   :1.0   Min.   :1.000
##  1st Qu.: 1600   1st Qu.:1.000   1st Qu.:1.0   1st Qu.:1.000
##  Median : 4480   Median :2.000   Median :2.0   Median :2.000
##  Mean   : 5345   Mean   :1.605   Mean   :1.8   Mean   :1.546
##  3rd Qu.: 7680   3rd Qu.:2.000   3rd Qu.:2.0   3rd Qu.:2.000
##  Max.   :25600   Max.   :2.000   Max.   :3.0   Max.   :2.000
##       AGE             PAY_1              PAY_2              PAY_3
##  Min.   :21.00   Min.   :-2.00000   Min.   :-2.0000   Min.   :-2.0000
##  1st Qu.:28.00   1st Qu.:-1.00000   1st Qu.:-1.0000   1st Qu.:-1.0000
##  Median :34.00   Median : 0.00000   Median : 0.0000   Median : 0.0000
##  Mean   :34.98   Mean   :-0.01714   Mean   :-0.1057   Mean   :-0.1401
##  3rd Qu.:41.00   3rd Qu.: 0.00000   3rd Qu.: 0.0000   3rd Qu.: 0.0000
##  Max.   :58.00   Max.   : 8.00000   Max.   : 8.0000   Max.   : 8.0000
##      PAY_4              PAY_5              PAY_6             BILL_AMT1.
##  Min.   :-2.0000   Min.   :-2.0000   Min.   :-2.0000   Min.   :-5298.6
```

```
##  1st Qu.:-1.0000   1st Qu.:-1.0000   1st Qu.:-1.0000   1st Qu.:  128.0
##  Median : 0.0000   Median : 0.0000   Median : 0.0000   Median :  744.1
##  Mean   :-0.1964   Mean   :-0.2423   Mean   :-0.2667   Mean   : 1649.9
##  3rd Qu.: 0.0000   3rd Qu.: 0.0000   3rd Qu.: 0.0000   3rd Qu.: 2167.2
##  Max.   : 8.0000   Max.   : 8.0000   Max.   : 8.0000   Max.   :23898.0
##    BILL_AMT2.         BILL_AMT3.         BILL_AMT4.
##  Min.   :-2232.9   Min.   :-5032.45   Min.   :-5440.00
##  1st Qu.:  109.4   1st Qu.:   99.31   1st Qu.:   84.79
##  Median :  706.4   Median :  658.21   Median :  619.33
##  Mean   : 1585.3   Mean   : 1516.71   Mean   : 1397.82
##  3rd Qu.: 2067.9   3rd Qu.: 1945.54   3rd Qu.: 1774.29
##  Max.   :23807.0   Max.   :53250.85   Max.   :22619.65
##    BILL_AMT5.          BILL_AMT6.          PAY_AMT1.
##  Min.   :-2602.69   Min.   :-10867.3   Min.   :    0.00
##  1st Qu.:   66.67   1st Qu.:    48.0   1st Qu.:   32.00
##  Median :  591.66   Median :   562.6   Median :   69.54
##  Mean   : 1304.60   Mean   :  1259.1   Mean   :  182.62
##  3rd Qu.: 1624.38   3rd Qu.:  1591.3   3rd Qu.:  160.90
##  Max.   :26353.28   Max.   : 22398.2   Max.   :16160.00
##    PAY_AMT2.           PAY_AMT3.           PAY_AMT4.
##  Min.   :    0.00   Min.   :    0.00   Min.   :    0.00
##  1st Qu.:   32.00   1st Qu.:   15.06   1st Qu.:   10.43
##  Median :   65.31   Median :   60.80   Median :   48.64
##  Mean   :  190.33   Mean   :  166.30   Mean   :  156.67
##  3rd Qu.:  160.00   3rd Qu.:  147.52   3rd Qu.:  131.20
##  Max.   :53896.29   Max.   :16263.33   Max.   :16924.70
##    PAY_AMT5.           PAY_AMT6.            def_pay
##  Min.   :    0.00   Min.   :    0.000   Min.   :0.0000
##  1st Qu.:   10.21   1st Qu.:    6.552   1st Qu.:0.0000
##  Median :   51.20   Median :   48.000   Median :0.0000
##  Mean   :  154.94   Mean   :  168.919   Mean   :0.2117
##  3rd Qu.:  133.77   3rd Qu.:  130.800   3rd Qu.:0.0000
##  Max.   :13648.93   Max.   :16917.312   Max.   :1.0000
```

xgb model before tuning

```r
xgb <- xgboost(data = data.matrix(train[,1:23]),
 label = data.matrix(y_tr),
 eta = 1,
 max_depth = 6,
 nround=300,
 subsample = 1,
 colsample_bytree = 1,
 lambda = 1,
 seed = 33,
 eval_metric = "error",
 objective = "binary:logistic",
)
```

```
## [1]  train-error:0.162800
## [2]  train-error:0.159100
## [3]  train-error:0.159000
```

```
## [4]  train-error:0.156550
## [5]  train-error:0.154600
## [6]  train-error:0.152550
## [7]  train-error:0.148650
## [8]  train-error:0.146700
## [9]  train-error:0.145150
## [10] train-error:0.144250
## [11] train-error:0.141250
## [12] train-error:0.139050
## [13] train-error:0.138000
## [14] train-error:0.134450
## [15] train-error:0.131650
## [16] train-error:0.129500
## [17] train-error:0.125400
## [18] train-error:0.121300
## [19] train-error:0.120500
## [20] train-error:0.117600
## [21] train-error:0.117200
## [22] train-error:0.116850
## [23] train-error:0.114050
## [24] train-error:0.113400
## [25] train-error:0.111900
## [26] train-error:0.111350
## [27] train-error:0.108850
## [28] train-error:0.106000
## [29] train-error:0.103400
## [30] train-error:0.100450
## [31] train-error:0.098450
## [32] train-error:0.096350
## [33] train-error:0.095950
## [34] train-error:0.095150
## [35] train-error:0.093050
## [36] train-error:0.092400
## [37] train-error:0.091750
## [38] train-error:0.090000
## [39] train-error:0.086150
## [40] train-error:0.083200
## [41] train-error:0.080800
## [42] train-error:0.079850
## [43] train-error:0.077900
## [44] train-error:0.076050
## [45] train-error:0.074050
## [46] train-error:0.072550
## [47] train-error:0.071300
## [48] train-error:0.069150
## [49] train-error:0.068050
## [50] train-error:0.066450
## [51] train-error:0.064750
## [52] train-error:0.063800
## [53] train-error:0.062100
## [54] train-error:0.061100
## [55] train-error:0.060150
## [56] train-error:0.057900
## [57] train-error:0.056400
```

```
## [58] train-error:0.054600
## [59] train-error:0.054100
## [60] train-error:0.052500
## [61] train-error:0.052750
## [62] train-error:0.050750
## [63] train-error:0.050200
## [64] train-error:0.049100
## [65] train-error:0.046900
## [66] train-error:0.045750
## [67] train-error:0.045000
## [68] train-error:0.044100
## [69] train-error:0.043650
## [70] train-error:0.042350
## [71] train-error:0.042100
## [72] train-error:0.040400
## [73] train-error:0.040350
## [74] train-error:0.038350
## [75] train-error:0.037800
## [76] train-error:0.037100
## [77] train-error:0.035900
## [78] train-error:0.035300
## [79] train-error:0.035150
## [80] train-error:0.034200
## [81] train-error:0.034100
## [82] train-error:0.033500
## [83] train-error:0.032600
## [84] train-error:0.031550
## [85] train-error:0.031100
## [86] train-error:0.030400
## [87] train-error:0.029550
## [88] train-error:0.028600
## [89] train-error:0.027800
## [90] train-error:0.027800
## [91] train-error:0.026950
## [92] train-error:0.025450
## [93] train-error:0.025050
## [94] train-error:0.023850
## [95] train-error:0.023000
## [96] train-error:0.022800
## [97] train-error:0.021650
## [98] train-error:0.021600
## [99] train-error:0.020200
## [100]    train-error:0.019750
## [101]    train-error:0.018600
## [102]    train-error:0.018300
## [103]    train-error:0.017750
## [104]    train-error:0.017400
## [105]    train-error:0.016700
## [106]    train-error:0.016200
## [107]    train-error:0.015900
## [108]    train-error:0.015450
## [109]    train-error:0.014250
## [110]    train-error:0.013400
## [111]    train-error:0.013150
```

```
## [112]	train-error:0.012450
## [113]	train-error:0.011600
## [114]	train-error:0.010950
## [115]	train-error:0.011050
## [116]	train-error:0.009900
## [117]	train-error:0.009250
## [118]	train-error:0.008800
## [119]	train-error:0.008150
## [120]	train-error:0.007500
## [121]	train-error:0.007400
## [122]	train-error:0.007250
## [123]	train-error:0.007450
## [124]	train-error:0.006750
## [125]	train-error:0.006800
## [126]	train-error:0.006950
## [127]	train-error:0.006050
## [128]	train-error:0.005700
## [129]	train-error:0.005450
## [130]	train-error:0.005350
## [131]	train-error:0.005000
## [132]	train-error:0.004600
## [133]	train-error:0.004650
## [134]	train-error:0.004500
## [135]	train-error:0.004300
## [136]	train-error:0.004200
## [137]	train-error:0.004250
## [138]	train-error:0.004050
## [139]	train-error:0.003950
## [140]	train-error:0.003800
## [141]	train-error:0.003750
## [142]	train-error:0.003750
## [143]	train-error:0.003350
## [144]	train-error:0.003150
## [145]	train-error:0.002900
## [146]	train-error:0.002700
## [147]	train-error:0.002250
## [148]	train-error:0.002300
## [149]	train-error:0.002000
## [150]	train-error:0.001900
## [151]	train-error:0.001650
## [152]	train-error:0.001550
## [153]	train-error:0.001700
## [154]	train-error:0.001500
## [155]	train-error:0.001500
## [156]	train-error:0.001400
## [157]	train-error:0.001400
## [158]	train-error:0.001350
## [159]	train-error:0.001300
## [160]	train-error:0.001300
## [161]	train-error:0.001300
## [162]	train-error:0.001300
## [163]	train-error:0.001200
## [164]	train-error:0.001200
## [165]	train-error:0.001200
```

```
## [166]	train-error:0.001050
## [167]	train-error:0.000950
## [168]	train-error:0.000950
## [169]	train-error:0.000900
## [170]	train-error:0.000850
## [171]	train-error:0.000750
## [172]	train-error:0.000750
## [173]	train-error:0.000750
## [174]	train-error:0.000800
## [175]	train-error:0.000800
## [176]	train-error:0.000850
## [177]	train-error:0.000750
## [178]	train-error:0.000800
## [179]	train-error:0.000800
## [180]	train-error:0.000750
## [181]	train-error:0.000750
## [182]	train-error:0.000750
## [183]	train-error:0.000700
## [184]	train-error:0.000650
## [185]	train-error:0.000600
## [186]	train-error:0.000650
## [187]	train-error:0.000650
## [188]	train-error:0.000650
## [189]	train-error:0.000550
## [190]	train-error:0.000550
## [191]	train-error:0.000350
## [192]	train-error:0.000350
## [193]	train-error:0.000400
## [194]	train-error:0.000400
## [195]	train-error:0.000400
## [196]	train-error:0.000400
## [197]	train-error:0.000250
## [198]	train-error:0.000300
## [199]	train-error:0.000100
## [200]	train-error:0.000250
## [201]	train-error:0.000250
## [202]	train-error:0.000250
## [203]	train-error:0.000200
## [204]	train-error:0.000150
## [205]	train-error:0.000150
## [206]	train-error:0.000200
## [207]	train-error:0.000150
## [208]	train-error:0.000150
## [209]	train-error:0.000100
## [210]	train-error:0.000100
## [211]	train-error:0.000150
## [212]	train-error:0.000150
## [213]	train-error:0.000150
## [214]	train-error:0.000150
## [215]	train-error:0.000150
## [216]	train-error:0.000150
## [217]	train-error:0.000150
## [218]	train-error:0.000150
## [219]	train-error:0.000150
```

```
## [220]	train-error:0.000150
## [221]	train-error:0.000150
## [222]	train-error:0.000150
## [223]	train-error:0.000150
## [224]	train-error:0.000150
## [225]	train-error:0.000150
## [226]	train-error:0.000150
## [227]	train-error:0.000150
## [228]	train-error:0.000150
## [229]	train-error:0.000150
## [230]	train-error:0.000100
## [231]	train-error:0.000100
## [232]	train-error:0.000150
## [233]	train-error:0.000150
## [234]	train-error:0.000150
## [235]	train-error:0.000150
## [236]	train-error:0.000150
## [237]	train-error:0.000150
## [238]	train-error:0.000150
## [239]	train-error:0.000150
## [240]	train-error:0.000100
## [241]	train-error:0.000100
## [242]	train-error:0.000100
## [243]	train-error:0.000100
## [244]	train-error:0.000150
## [245]	train-error:0.000100
## [246]	train-error:0.000100
## [247]	train-error:0.000100
## [248]	train-error:0.000100
## [249]	train-error:0.000100
## [250]	train-error:0.000100
## [251]	train-error:0.000100
## [252]	train-error:0.000100
## [253]	train-error:0.000100
## [254]	train-error:0.000100
## [255]	train-error:0.000100
## [256]	train-error:0.000100
## [257]	train-error:0.000100
## [258]	train-error:0.000100
## [259]	train-error:0.000100
## [260]	train-error:0.000100
## [261]	train-error:0.000100
## [262]	train-error:0.000100
## [263]	train-error:0.000100
## [264]	train-error:0.000100
## [265]	train-error:0.000100
## [266]	train-error:0.000100
## [267]	train-error:0.000100
## [268]	train-error:0.000100
## [269]	train-error:0.000100
## [270]	train-error:0.000100
## [271]	train-error:0.000100
## [272]	train-error:0.000100
## [273]	train-error:0.000100
```

```
## [274]    train-error:0.000100
## [275]    train-error:0.000100
## [276]    train-error:0.000100
## [277]    train-error:0.000100
## [278]    train-error:0.000100
## [279]    train-error:0.000100
## [280]    train-error:0.000100
## [281]    train-error:0.000100
## [282]    train-error:0.000100
## [283]    train-error:0.000100
## [284]    train-error:0.000100
## [285]    train-error:0.000100
## [286]    train-error:0.000100
## [287]    train-error:0.000100
## [288]    train-error:0.000100
## [289]    train-error:0.000100
## [290]    train-error:0.000100
## [291]    train-error:0.000100
## [292]    train-error:0.000100
## [293]    train-error:0.000100
## [294]    train-error:0.000100
## [295]    train-error:0.000100
## [296]    train-error:0.000100
## [297]    train-error:0.000100
## [298]    train-error:0.000100
## [299]    train-error:0.000100
## [300]    train-error:0.000100
```

result before tuning on test dataset, output is accuracy rate

```r
y_test <- predict(xgb, data.matrix(test[,1:23]))
y_test[y_test < 0.5] = 0
y_test[y_test > 0.5] = 1
1 - sum(y_test == y_ts)/length(y_ts)
```

```
## [1] 0.2023923
```

```r
y_prediction = y_test
tbl = table(y_prediction, y_ts)
print(tbl)
```

```
##             y_ts
## y_prediction    0    1
##            0 5965 1023
##            1  669  703
```

```r
print(paste("Precision: ", tbl[2,2]/sum(tbl[2,])))
```

```
## [1] "Precision:  0.512390670553936"
```

```r
print(paste("Recall: ", tbl[2,2]/sum(tbl[,2])))
```

```
## [1] "Recall:  0.407300115874855"
```

```r
library(caret)
precision <- posPredValue(as.factor(y_prediction), as.factor(y_ts), positive="1")
recall <- sensitivity(as.factor(y_prediction), as.factor(y_ts), positive="1")
print(paste("Precision: ", precision))
```

```
## [1] "Precision:  0.512390670553936"
```

```r
print(paste("Recall: ", recall))
```

```
## [1] "Recall:  0.407300115874855"
```

xgb model after tuning

```r
xgb <- xgboost(data = data.matrix(train[,1:23]),
 label = data.matrix(y_tr),
 eta = 0.05,
 max_depth = 3,
 nround=10,
 subsample = 1,
 colsample_bytree = 1,
 lambda = 10,
 seed = 33,
 eval_metric = "error",
 objective = "binary:logistic",
)
```

```
## [1]  train-error:0.169300
## [2]  train-error:0.169150
## [3]  train-error:0.169300
## [4]  train-error:0.169150
## [5]  train-error:0.169150
## [6]  train-error:0.169150
## [7]  train-error:0.169150
## [8]  train-error:0.169100
## [9]  train-error:0.169100
## [10] train-error:0.169150
```

result after tuning on test dataset, output is accuracy rate

```r
y_test <- predict(xgb, data.matrix(test[,1:23]))
y_test[y_test < 0.5] = 0
y_test[y_test > 0.5] = 1
1 - sum(y_test == y_ts)/length(y_ts)
```

```
## [1] 0.1601675
```

```r
y_prediction = y_test
tbl = table(y_prediction, y_ts)
print(tbl)
```

```
##             y_ts
## y_prediction    0    1
##            0 6282  987
##            1  352  739
```

```r
print(paste("Precision: ", tbl[2,2]/sum(tbl[2,])))
```

```
## [1] "Precision:  0.677360219981668"
```

```r
print(paste("Recall: ", tbl[2,2]/sum(tbl[,2])))
```

```
## [1] "Recall:  0.428157589803013"
```

```r
precision <- posPredValue(as.factor(y_prediction), as.factor(y_ts), positive="1")
recall <- sensitivity(as.factor(y_prediction), as.factor(y_ts), positive="1")
print(paste("Precision: ", precision))
```

```
## [1] "Precision:  0.677360219981668"
```

```r
print(paste("Recall: ", recall))
```

```
## [1] "Recall:  0.428157589803013"
```

feature importance

Gain

is the improvement in accuracy brought by a feature to the branches it is on. The idea is that before adding a new split on a feature X to the branch there was some wrongly classified elements, after adding the split on this feature, there are two new branches, and each of these branch is more accurate (one branch saying if your observation is on this branch then it should be classified as 1, and the other branch saying the exact opposite).

Cover

measures the relative quantity of observations concerned by a feature.

Frequency

is a simpler way to measure the Gain. It just counts the number of times a feature is used in all generated trees. You should not use it (unless you know why you want to use it).

```r
xgb.importance(model = xgb)
```

```
##        Feature         Gain        Cover  Frequency
##  1:      PAY_1 0.7634134859 0.3333333306 0.14285714
##  2:      PAY_2 0.1660897340 0.2980205198 0.14285714
##  3:      PAY_4 0.0181468821 0.1097148970 0.05714286
##  4: LIMIT_BAL. 0.0163464585 0.1074236377 0.05714286
##  5:      PAY_3 0.0136124543 0.0522952944 0.14285714
##  6:      PAY_6 0.0095069087 0.0372762791 0.18571429
##  7:      PAY_5 0.0050551374 0.0374183922 0.05714286
##  8: BILL_AMT1. 0.0025109365 0.0104823072 0.04285714
##  9:  PAY_AMT4. 0.0020528418 0.0008665749 0.05714286
## 10: BILL_AMT5. 0.0015292959 0.0006500527 0.04285714
## 11:  PAY_AMT6. 0.0011834776 0.0091800985 0.04285714
## 12:  PAY_AMT5. 0.0003123546 0.0032171827 0.01428571
## 13:        AGE 0.0002400327 0.0001214331 0.01428571
```

CV: iterates by nrounds

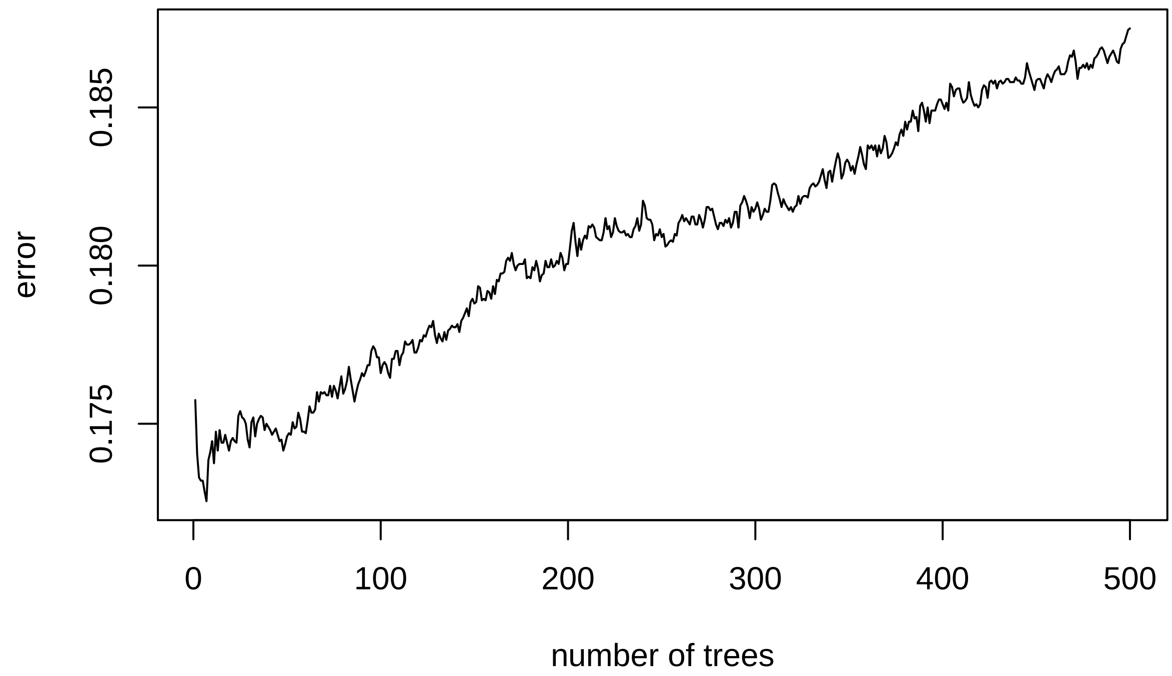
```r
nrounds = 500

credit = xgb.DMatrix(as.matrix(train[,1:23]), label = as.matrix(y_tr))

cv <- xgb.cv(data = credit, nrounds = nrounds, nfold = 5, metrics = "error",
                  max_depth = 6, eta = 0.3, objective = "binary:logistic",
                subsample = 1, colsample_bytree = 1, seed = 33,
                lambda = 20, verbose = 0)
print(cv)
```

```
## ##### xgb.cv 5-folds
##     iter train_error_mean train_error_std test_error_mean test_error_std
##        1        0.1606376    0.0012086863         0.17575    0.004753946
##        2        0.1611374    0.0016680835         0.17405    0.004374357
##        3        0.1607254    0.0018173318         0.17330    0.003014133
##        4        0.1602250    0.0017976478         0.17320    0.004154516
##        5        0.1600252    0.0014320623         0.17320    0.003472751
## ---
##      496        0.0180752    0.0010879718         0.18700    0.005272571
##      497        0.0179124    0.0009332176         0.18705    0.005122011
##      498        0.0179250    0.0009465134         0.18725    0.004954796
##      499        0.0177872    0.0010235588         0.18745    0.005119570
##      500        0.0178126    0.0009584493         0.18750    0.005203364
```

visualize CV result: we don't really need a lot of trees

Conclusion: 10 trees would work ok

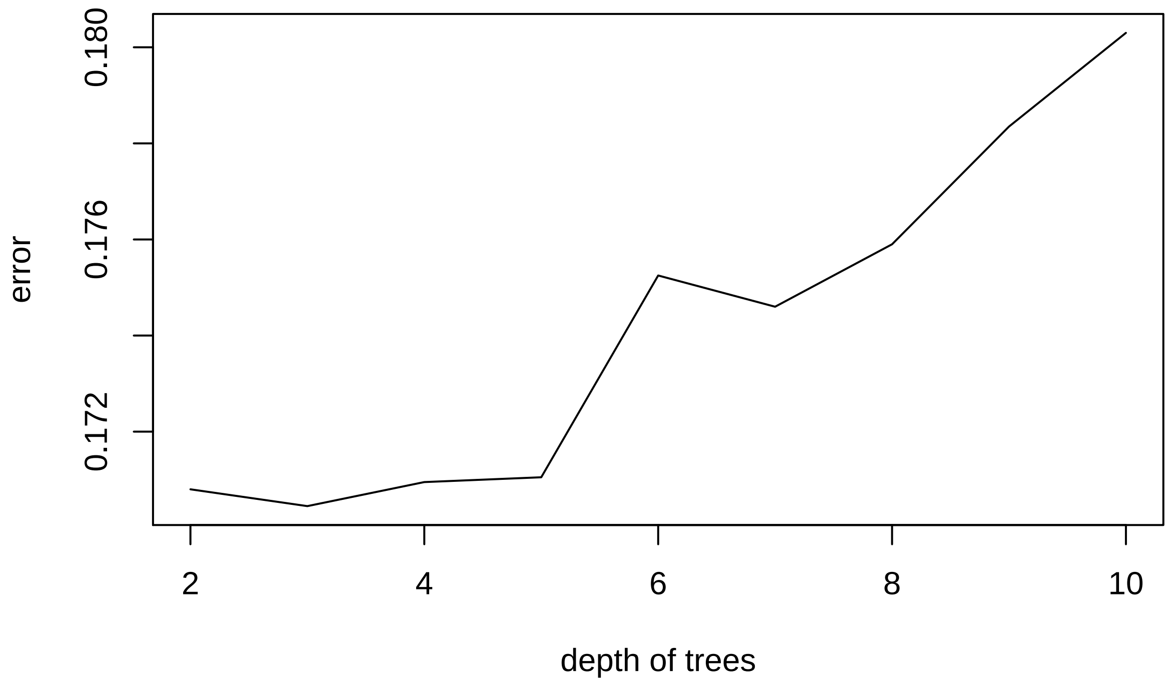
```r
plot(c(1:nrounds), cv[4]$evaluation_log$test_error_mean, xlab = 'number of trees', ylab = 'error', type
```



how does the depth of each tree affects error

Conclusion: 3 splits for each tree seems to be optimized

```
start_time <- Sys.time()
minError = c()

for(i in c(2:10)){
        cv <- xgb.cv(data = credit, nrounds = 10, nfold = 5, metrics = "error",
                  max_depth = i, eta = 0.3, objective = "binary:logistic",
                subsample = 1, colsample_bytree = 1, seed = 33,
                lambda = 1, verbose = 0)
        minError = c(minError, min(cv[4]$evaluation_log$test_error_mean))
}

plot(c(2:10), minError, xlab = 'depth of trees', ylab = 'error', type = 'l')
```



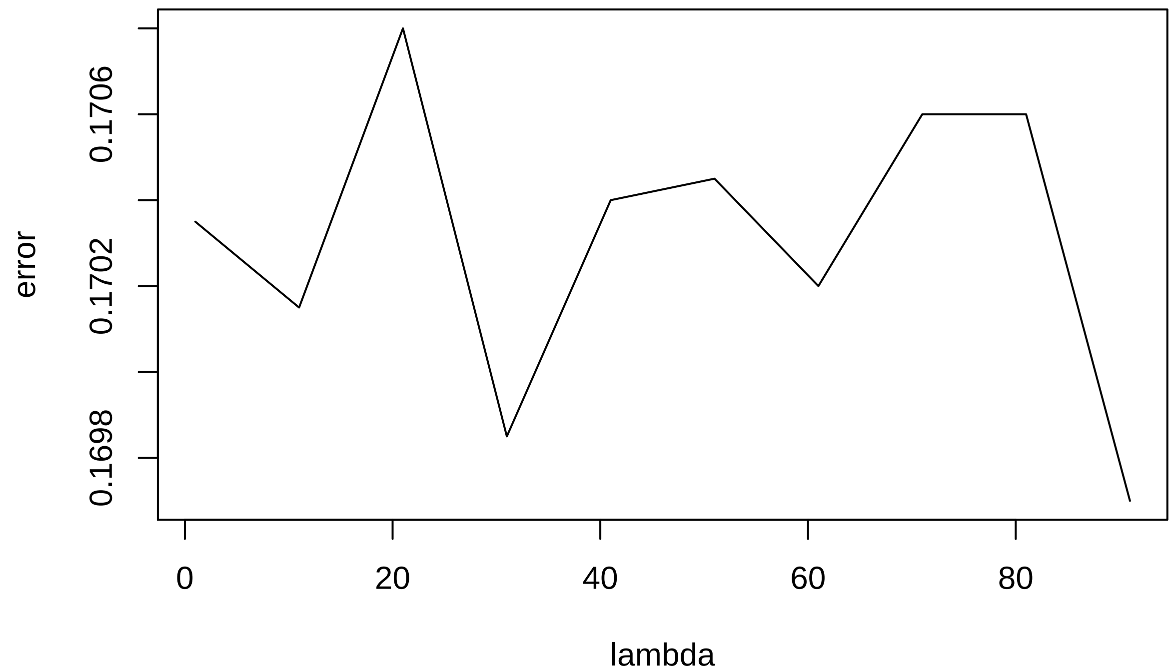

```
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 7.634531 secs
```

how lambda affects error

conclusion: lambda 10 would work ok

```
start_time <- Sys.time()
testVariable = 'lambda'
testRange = seq(1, 100, 10)
minError = c()

for(i in testRange){
        cv <- xgb.cv(data = credit, nrounds = 10, nfold = 5, metrics = "error",
                  max_depth = 3, eta = 0.3, objective = "binary:logistic",
                subsample = 1, colsample_bytree = 1, seed = 33,
                lambda = i, verbose = 0)
        minError = c(minError, min(cv[4]$evaluation_log$test_error_mean))
}

plot(testRange, minError, xlab = testVariable, ylab = 'error', type = 'l')
```

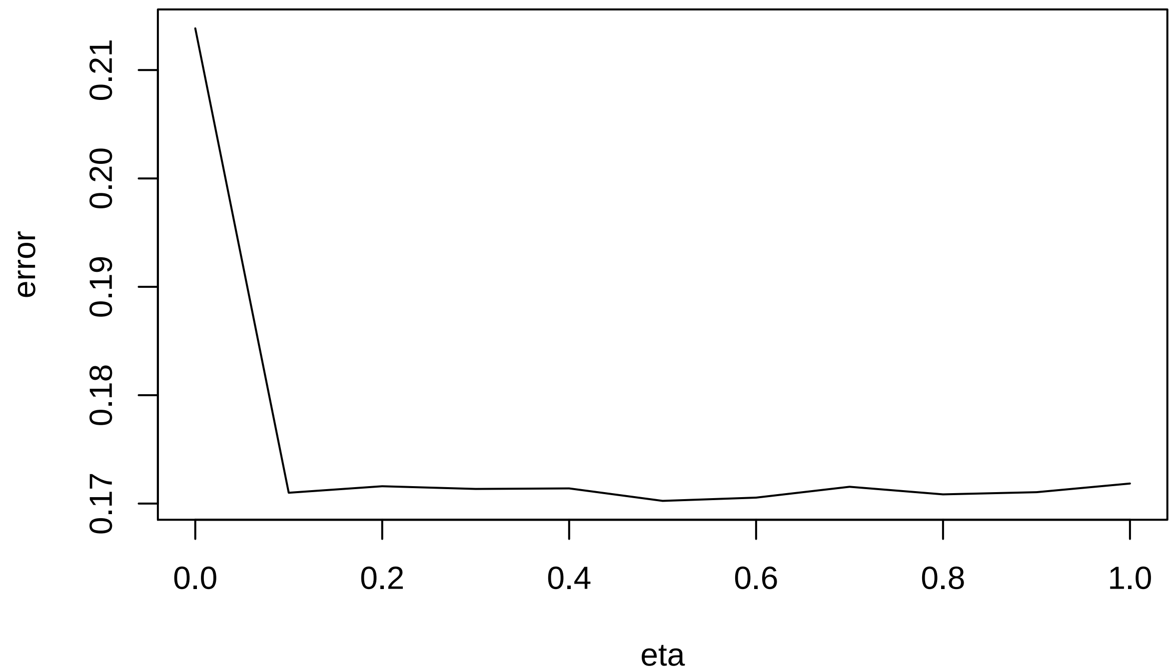



```
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 3.463929 secs
```

how eta affects error

```
start_time <- Sys.time()
testVariable = 'eta'
testRange = seq(0.0, 1, 0.1)
minError = c()

for(i in testRange){
        cv <- xgb.cv(data = credit, nrounds = 10, nfold = 5, metrics = "error",
                  max_depth = 3, eta = i, objective = "binary:logistic",
                subsample = 0.8, colsample_bytree = 0.8, seed = 33,
                lambda = 10, verbose = 0)
        minError = c(minError, min(cv[4]$evaluation_log$test_error_mean))
}

plot(testRange, minError, xlab = testVariable, ylab = 'error', type = 'l')
```




```
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 4.285434 secs
```