Writing:
Code:
Analysis:
Results discussed by:

Part 1

ctrl alt i for r code field

```
set.seed(12345)
library(knitr)
library(dplyr)
```

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

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

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

```
library(tidyr)
data = read.csv("bank-full.csv", header=TRUE, sep=";")
data = select(data, -duration)

#split data into 40/30/30 training/test/validation, code from lecture
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.4))
train=data[id,]
id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.3))
valid=data[id2,]
id3=setdiff(id1,id2)
test=data[id3,]
```

Part 2

```
library(tree)
set.seed(12345)

cf_matrix <- function(tree, data, y) {

  Probs = predict(tree, newdata = data)
  bestI = apply(Probs, MARGIN=1, FUN = which.max)
  Pred=levels(y)[bestI]
  cf_matrix = table(y, Pred)
  return(cf_matrix)
}

miss_rate = function(cf_matrix) {
  miss_rate = sum(rowSums(cf_matrix) - diag(cf_matrix)) / sum(cf_matrix)
  return(miss_rate)
}
```

```r
#a. Decision Tree with default settings.
tree_default = tree(as.factor(y)~., data=train)
cf_def_train = cf_matrix(tree_default, train, train$y)
cf_def_valid = cf_matrix(tree_default, valid, valid$y)
mr_def_train = miss_rate(cf_def_train)
mr_def_valid = miss_rate(cf_def_valid)

print("Tree with default settings")
```

```
## [1] "Tree with default settings"
```

```r
print(paste("Miss rate train:", mr_def_train))
```

```
## [1] "Miss rate train: 0.104844061048441"
```

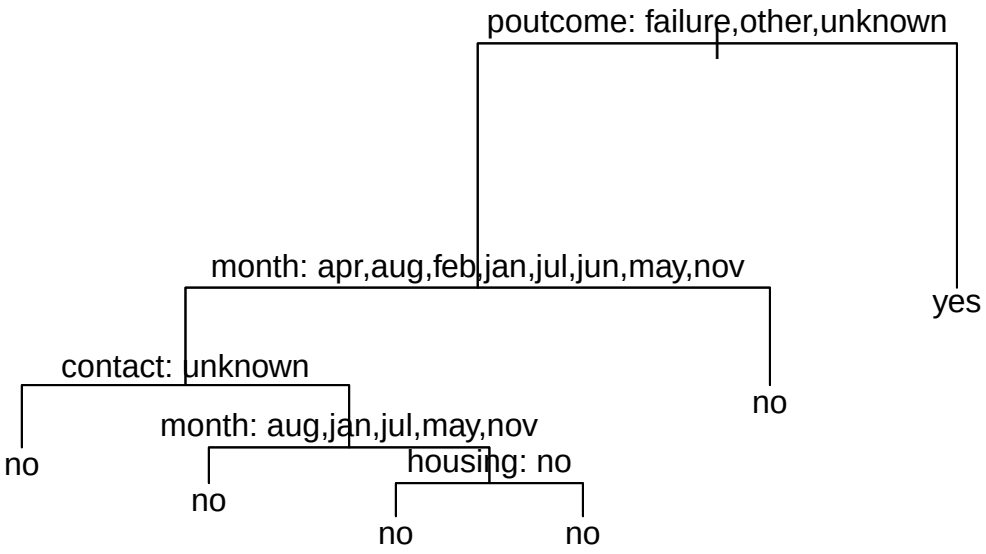
```r
print(paste("Miss rate valid:", mr_def_valid))
```

```
## [1] "Miss rate valid: 0.109267861092679"
```

```r
summary(tree_default)
```

```
##
## Classification tree:
## tree(formula = as.factor(y) ~ ., data = train)
## Variables actually used in tree construction:
## [1] "poutcome" "month"    "contact"  "housing"
## Number of terminal nodes:  6
## Residual mean deviance:  0.6022 = 10890 / 18080
## Misclassification error rate: 0.1048 = 1896 / 18084
```

```r
plot(tree_default)
text(tree_default, pretty=0)
```




```
#b. Decision Tree with smallest allowed node size equal to 7000
tree_7000 = tree(as.factor(y)~., data=train, control = tree.control(nrow(data), minsize = 7000))
cf_7000_train = cf_matrix(tree_7000, train, train$y)
cf_7000_valid = cf_matrix(tree_7000, valid, valid$y)
mr_7000_train = miss_rate(cf_7000_train)
mr_7000_valid = miss_rate(cf_7000_valid)

print("Tree with smallest allowed node size 7000")
```

```
## [1] "Tree with smallest allowed node size 7000"
```

```
print(paste("Miss rate train:", mr_7000_train))
```

```
## [1] "Miss rate train: 0.104844061048441"
```

```
print(paste("Miss rate valid:", mr_7000_valid))
```

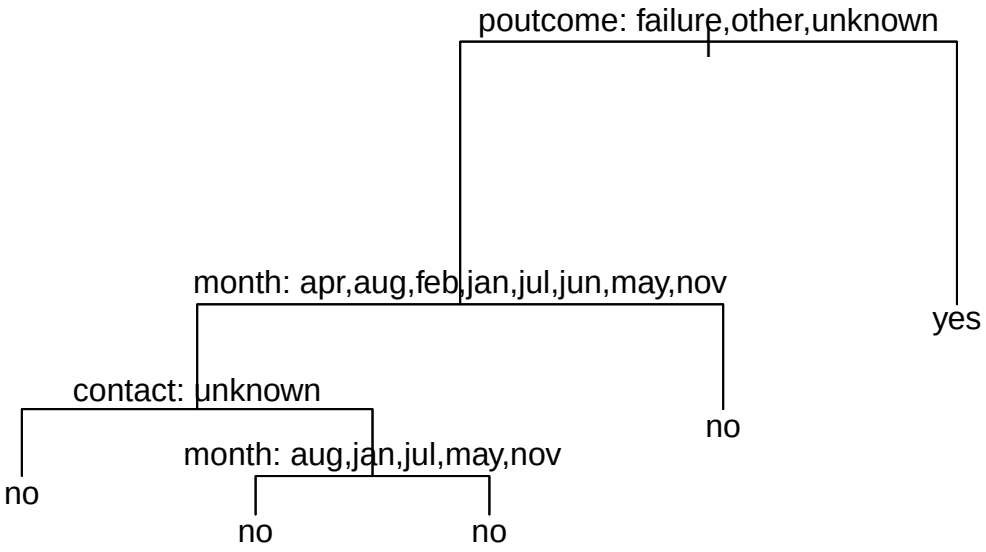
```
## [1] "Miss rate valid: 0.109267861092679"
```

```
summary(tree_7000)
```

```
##
## Classification tree:
## tree(formula = as.factor(y) ~ ., data = train, control = tree.control(nrow(data),
##     minsize = 7000))
## Variables actually used in tree construction:
## [1] "poutcome" "month"    "contact"
## Number of terminal nodes:  5
## Residual mean deviance:  0.6097 = 11020 / 18080
```

```
## Misclassification error rate: 0.1048 = 1896 / 18084
```

```
plot(tree_7000)
text(tree_7000, pretty=0)
```




```
#c. Decision trees minimum deviance to 0.0005
tree_0005 = tree(as.factor(y)~., data=train, control = tree.control(nrow(data), mindev = 0.0005))
cf_0005_train = cf_matrix(tree_0005, train, train$y)
cf_0005_valid = cf_matrix(tree_0005, valid, valid$y)
mr_0005_train = miss_rate(cf_0005_train)
mr_0005_valid = miss_rate(cf_0005_valid)

print("Tree with smallest deviance of 0.0005")
```

```
## [1] "Tree with smallest deviance of 0.0005"
```

```
print(paste("Miss rate train:", mr_0005_train))
```

```
## [1] "Miss rate train: 0.0936186684361867"
```

```
print(paste("Miss rate valid:", mr_0005_valid))
```

```
## [1] "Miss rate valid: 0.11170095111701"
```

```
summary(tree_0005)
```

```
##
## Classification tree:
## tree(formula = as.factor(y) ~ ., data = train, control = tree.control(nrow(data),
```

```
##     mindev = 5e-04))
## Variables actually used in tree construction:
##  [1] "poutcome"  "month"     "contact"   "marital"   "day"       "campaign"
##  [7] "job"       "pdays"     "age"       "balance"   "housing"   "education"
## [13] "previous"
## Number of terminal nodes:  122
## Residual mean deviance:  0.5213 = 9363 / 17960
## Misclassification error rate: 0.09362 = 1693 / 18084
```

```
plot(tree_0005)
```

```
#text(tree_0005, pretty=0)
```

By looking at the misclassification rates for the validation data, we can see that the trees with the default settings and the tree with minimum leaf node size of 7000 had the same lowest misclassification rate. Interestingly, the tree size of those two differed, but provided the same result. This makes sense if we plot the decision trees; all outcomes to the left of the *poutcome* node will result in a *no* and therefore, it does not matter if we add more nodes as long as all possible outcomes are *no*.

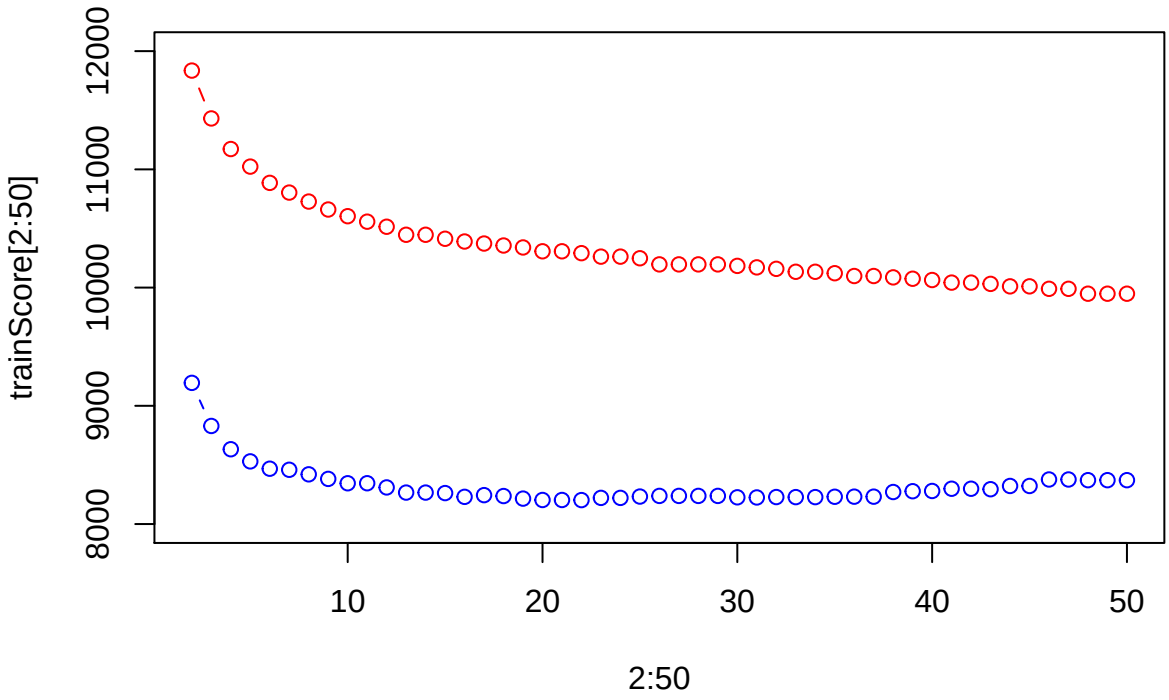
The tree with the default settings had 6 leaf nodes. When changing the minimum leaf node size to 7000 the tree size decreased to 5 nodes. When the minimum deviance was set to 0.005, the tree size increased significantly to a size of 122 nodes. This makes sense, by setting a higher minimum node size, we allow the nodes to split less. On the contrary, by lowering the minimum deviance (the child node's deviance needs to be at least the deviance of the parent multiplied by the set minimum deviance value in order to be allowed to split), we allow the nodes to split more. Since this significantly bigger tree performed worse the the other two, we can conclude that the model was probably overfitted.

Part 3

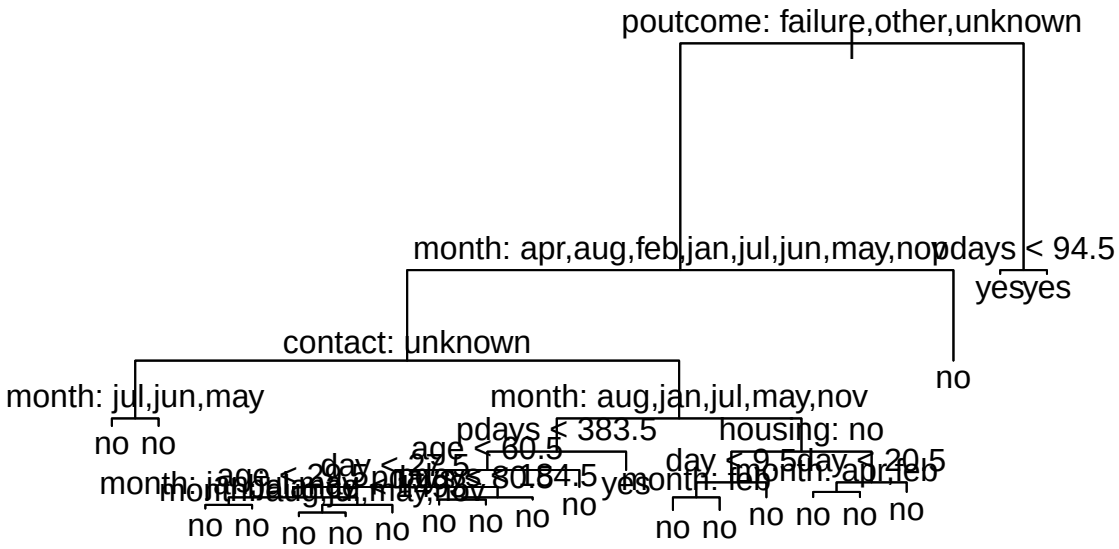
```
#code from lecture with some modifications
set.seed(12345)
trainScore=rep(0,50)
testScore=rep(0,50)
for(i in 2:50) {
prunedTree=prune.tree(tree_0005,best=i)
pred=predict(prunedTree, newdata=valid,
type="tree")
trainScore[i]=deviance(prunedTree)
testScore[i]=deviance(pred)
}
which.min(testScore[2:50])
```

```
## [1] 21
```

```
plot(2:50, trainScore[2:50], type="b", col="red",
ylim=c(8000,12000))
points(2:50, testScore[2:50], type="b", col="blue")
```




```
tree_optimum = prune.tree(tree_0005, best=21)
plot(tree_optimum)
text(tree_optimum, pretty=0)
```




```
summary(tree_optimum)
```

```
##
## Classification tree:
## snip.tree(tree = tree_0005, nodes = c(581L, 17L, 577L, 79L, 37L,
## 77L, 576L, 153L, 580L, 6L, 1157L, 16L, 5L, 1156L, 156L, 152L,
## 579L, 7L))
## Variables actually used in tree construction:
## [1] "poutcome" "month"    "contact"  "pdays"    "age"      "day"      "balance"
## [8] "housing"
## Number of terminal nodes:  21
## Residual mean deviance:  0.5706 = 10310 / 18060
## Misclassification error rate: 0.1041 = 1882 / 18084
```

By looking at the minimum score for the deviance in the validation data, we can determine that the tree with 21 leaf nodes had the lowest deviance. By looking at the graph, we can conclude that models with fewer leaf nodes perform worse since they have high bias and therefore tend to not capture all the trends in the training data (underfitting). On the other hand, after a certain threshhold, when the number of leaf nodes increase, the model will start to perform worse since it is overfitted to the training data. This means that the model has low bias (since it predicts the training data well) but high variance as it does not predict future data sets with the same accuracy, or in other words, the model is not very flexible. The model with 21 leaf nodes offers a good tradeoff between the metrics *bias* and *variance* and therefore performs the best.

In all of the models, the variable *poutcome* (the outcome of previous marketing campaigns) is the most important since it is the first decision point in the tree. The non-statistical interpretation of this is that it intuitively makes sense that previous marketing campaigns would predict the outcomes of similar marketing campaigns in the future. Other important variables (based on their position in the tree) are: *month,* and

contact, though we would argue that they are not really that important since their outcomes do not matter for the model since all leaf nodes to the left of *poutcome* result in a *no.*

Part 4

```
cf_matrix_opt = cf_matrix(tree_optimum, test, test$y)
cf_matrix_opt
```

```
##      Pred
## y        no   yes
##   no  11812   167
##   yes  1294   291
```

```
#code from tutorial with minor modifications
TP= cf_matrix_opt[2,2]
TN= cf_matrix_opt[1,1]
FP= cf_matrix_opt[1,2]
FN= cf_matrix_opt[2,1]
TPR=TP/(TP+FN)
FPR=FP/(FP+TN)
prec=TP/(TP+FP)
rec=TP/(TP+FN)

f1 = (2*prec*rec) / (prec+rec)
f1
```

```
## [1] 0.2848752
```

```
miss_rate_opt = miss_rate(cf_matrix_opt)
miss_rate_opt
```

```
## [1] 0.1077116
```

The F1 score is useful in this case since the *no* class is much larger than the *yes* class and therefore, just by predicting *no,* it could seem at first glance that the model performs well when the "good" prediction rate in fact comes from the class imbalance. In this case, if we were to predict no all the time, we would get a misclassification rate of 0.108. However, the F1 score tells a different story because it takes the class imbalance in consideration. We got a score of 0.28 which indicates that our model in fact performs poorly.

Part 5

```
set.seed(12345)
Probs = predict(tree_optimum, newdata = test)
Losses=Probs%*%matrix(c(0,5,1,0), nrow=2)
bestI=apply(Losses, MARGIN=1, FUN = which.min)
Pred=levels(test$y)[bestI]
cf_loss = table(test$y, Pred)
cf_loss
```

```
##      Pred
##          no   yes
##   no  11030   949
##   yes   771   814
```

```
TP= cf_loss[2,2]
TN= cf_loss[1,1]
FP= cf_loss[1,2]
FN= cf_loss[2,1]
```

```
prec=TP/(TP+FP)
rec=TP/(TP+FN)

f1 = (2*prec*rec) / (prec+rec)
f1
```

```
## [1] 0.4862605
```

```
miss_rate_loss = miss_rate(cf_loss)
miss_rate_loss
```

```
## [1] 0.1268063
```

By adding the loss matrix, we can weight the prediction so certain outcomes are avoided by the model e.g., in screening for diseases, the worst outcome would be a false negative, the consequences for that would be far worse than a false positive. In this model, we weight it so that it is five times worse to give a false positive compared to a false negative. By doing this, the F1 score increased significantly to 0.486, however the overall accuracy decreased a little (the misclassification rate increased to 0.127). Depending on the application (as described above), this could be a beneficial trade-off.

Part 6

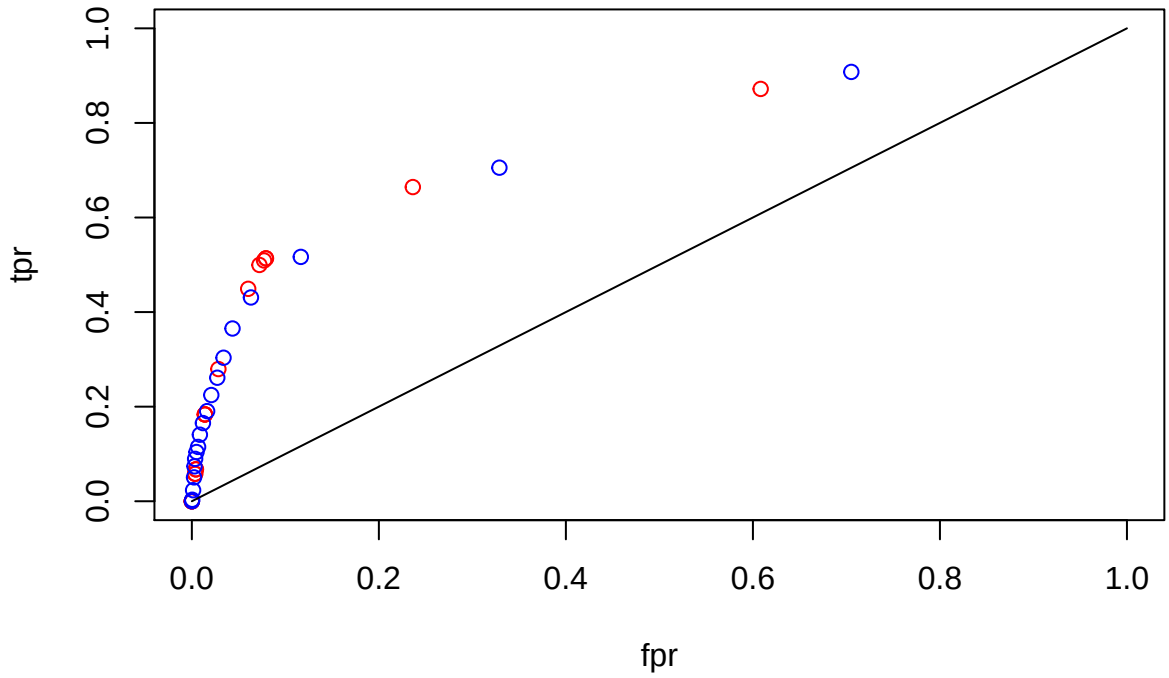
```
set.seed(12345)
pi = seq(0.05,0.95,0.05)
# Store TPR, FPR for each row
#ROCs_tree = matrix(1,2, length(pi))
#ROCs_log = matrix(1,2, length(pi))
tpr_tree = matrix(0, nrow=1, ncol=length(pi))
fpr_tree = matrix(0, nrow=1, ncol=length(pi))

tpr_log = matrix(0, nrow=1, ncol=length(pi))
fpr_log = matrix(0, nrow=1, ncol=length(pi))

#logistic regression
m2 = glm(y~., data=train, family = "binomial")
m2_pred = predict(m2, test, type='response')
#Probs = predict(tree_optimum, newdata = test)

for (i in 1:length(pi)) {
  pred_tree = ifelse(Probs[,2]>pi[i], "yes", "no")
  pred_log = ifelse(m2_pred>pi[i], "yes", "no")

  cf_tree = table(pred_tree, test$y)
  #print(cf_tree)
  # sometimes cf_tree has only one row and then this code will throw errors
  if (length(cf_tree[,1])>1) {
    tp_tree = cf_tree[2,2]
    fn_tree = cf_tree[1,2]
    fp_tree = cf_tree[2,1]
    tn_tree = cf_tree[1,1]

    tpr_tree[i] = tp_tree/(tp_tree+fn_tree)
    fpr_tree[i] = fp_tree/(fp_tree+tn_tree)
  }

  cf_log = table(pred_log, test$y)
```

```
  print(cf_log)
  tp_log = cf_log[2,2]
  fn_log = cf_log[1,2]
  fp_log = cf_log[2,1]
  tn_log = cf_log[1,1]

  tpr_log[i] = tp_log/(tp_log+fn_log)
  fpr_log[i] = fp_log/(fp_log+tn_log)



}
```

```
##
## pred_log   no  yes
##      no  3531  146
##      yes 8448 1439
##
## pred_log   no  yes
##      no  8039  467
##      yes 3940 1118
##
## pred_log    no   yes
##      no  10583   766
##      yes  1396   819
##
## pred_log    no   yes
##      no  11222   902
##      yes   757   683
##
## pred_log    no   yes
##      no  11458  1006
##      yes   521   579
##
## pred_log    no   yes
##      no  11573  1104
##      yes   406   481
##
## pred_log    no   yes
##      no  11653  1171
##      yes   326   414
##
## pred_log    no   yes
##      no  11730  1229
##      yes   249   356
##
## pred_log    no   yes
##      no  11783  1283
##      yes   196   302
##
## pred_log    no   yes
##      no  11837  1323
##      yes   142   262
##
```

```
## pred_log    no   yes
##      no  11876  1362
##      yes   103   223
##
## pred_log    no   yes
##      no  11899  1403
##      yes    80   182
##
## pred_log    no   yes
##      no  11919  1420
##      yes    60   165
##
## pred_log    no   yes
##      no  11937  1443
##      yes    42   142
##
## pred_log    no   yes
##      no  11945  1468
##      yes    34   117
##
## pred_log    no   yes
##      no  11951  1505
##      yes    28    80
##
## pred_log    no   yes
##      no  11963  1548
##      yes    16    37
##
## pred_log    no   yes
##      no  11974  1580
##      yes     5     5
##
## pred_log    no   yes
##      no  11978  1585
##      yes     1     0
```

```
#ROC or Receiver Operating Characteristic curve is used to evaluate logistic regression classification
x_grid=seq(0,1,0.01)
plot(fpr_tree, tpr_tree, col="red", xlim=c(0,1), ylim=c(0,1), xlab="fpr", ylab="tpr")
points(fpr_log, tpr_log, col="blue")
lines(x_grid, x_grid)
```




```
print("tree")
```

```
## [1] "tree"
```

```
print(paste("tpr tree: ", tpr_tree))
```

```
##  [1] "tpr tree:  0.87192429022082"   "tpr tree:  0.664353312302839"
##  [3] "tpr tree:  0.513564668769716"  "tpr tree:  0.513564668769716"
##  [5] "tpr tree:  0.509148264984227"  "tpr tree:  0.499684542586751"
##  [7] "tpr tree:  0.449211356466877"  "tpr tree:  0.279495268138801"
##  [9] "tpr tree:  0.183596214511041"  "tpr tree:  0.183596214511041"
## [11] "tpr tree:  0.183596214511041"  "tpr tree:  0.0675078864353312"
## [13] "tpr tree:  0.0675078864353312" "tpr tree:  0.0580441640378549"
## [15] "tpr tree:  0.0580441640378549" "tpr tree:  0"
## [17] "tpr tree:  0"                  "tpr tree:  0"
## [19] "tpr tree:  0"
```

```
print(paste("fpr tree: ", fpr_tree))
```

```
##  [1] "fpr tree:  0.608314550463311"   "fpr tree:  0.236330244594707"
##  [3] "fpr tree:  0.0792219717839553"  "fpr tree:  0.0792219717839553"
##  [5] "fpr tree:  0.0772184656482177"  "fpr tree:  0.0722931797311963"
##  [7] "fpr tree:  0.060272142916771"   "fpr tree:  0.0283830035896152"
##  [9] "fpr tree:  0.0139410635278404"  "fpr tree:  0.0139410635278404"
## [11] "fpr tree:  0.0139410635278404"  "fpr tree:  0.00434092996076467"
## [13] "fpr tree:  0.00434092996076467" "fpr tree:  0.0035061357375407"
## [15] "fpr tree:  0.0035061357375407"  "fpr tree:  0"
## [17] "fpr tree:  0"                   "fpr tree:  0"
```

```
## [19] "fpr tree:  0"
```

```
print("log")
```

```
## [1] "log"
```

```
print(paste("tpr log: ", tpr_log))
```

```
##  [1] "tpr log:  0.90788643533123"    "tpr log:  0.705362776025237"
##  [3] "tpr log:  0.516719242902208"   "tpr log:  0.430914826498423"
##  [5] "tpr log:  0.365299684542587"   "tpr log:  0.303470031545741"
##  [7] "tpr log:  0.261198738170347"   "tpr log:  0.224605678233438"
##  [9] "tpr log:  0.190536277602524"   "tpr log:  0.165299684542587"
## [11] "tpr log:  0.140694006309148"   "tpr log:  0.114826498422713"
## [13] "tpr log:  0.10410094637224"    "tpr log:  0.089589905362776"
## [15] "tpr log:  0.0738170347003155"  "tpr log:  0.0504731861198738"
## [17] "tpr log:  0.0233438485804416"  "tpr log:  0.00315457413249211"
## [19] "tpr log:  0"
```

```
print(paste("fpr log: ", fpr_log))
```

```
##  [1] "fpr log:  0.705234159779614"    "fpr log:  0.328908923950246"
##  [3] "fpr log:  0.116537273562067"    "fpr log:  0.0631939226980549"
##  [5] "fpr log:  0.0434927790299691"   "fpr log:  0.0338926454628934"
##  [7] "fpr log:  0.0272142916771016"   "fpr log:  0.020786376158277"
##  [9] "fpr log:  0.0163619667751899"   "fpr log:  0.0118540779697804"
## [11] "fpr log:  0.00859838049920695"  "fpr log:  0.0066783537857918"
## [13] "fpr log:  0.00500876533934385"  "fpr log:  0.0035061357375407"
## [15] "fpr log:  0.00283830035896152"  "fpr log:  0.00233742382502713"
## [17] "fpr log:  0.00133567075715836"  "fpr log:  0.000417397111611988"
## [19] "fpr log:  8.34794223223975e-05"
```

In this assignment, we have calculated the *True Positive Rate* and *False positive rate* for the logistic regression model and the optimal tree from previous exercises. In the *Receiver Operating Characteristic curve (ROC),* we can see that the optimum tree model (red) performs slightly better than the logistic regression model (blue). It can also be concluded that both models can be considered as "good" since they are well above the baseline (y=x) since that would be when the model predicts values randomly.

In this case, since there is a big class imbalance for many of the confusion matrices, it would be more appropriate to evaluate the models with a precision-recall curve instead of a ROC.