hw2

Victoria Xie

4/7/2022

Part 2

1(a)

```r
# read in data
Perdue1 <- 'immigration aliens criminals country loophole'
Perdue2 <- 'voter economy tax jobs security'
Perdue3 <- 'healthcare cost socialism unfair help'

Ossoff1 <- 'immigration country diversity help security'
Ossoff2 <- 'healthcare affordable expansion unfair help'
Ossoff3 <- 'voter relief debt jobs help'
Ossoff4 <- 'abortion choice right women help court'

emails <- data.frame(
  sender = c("Perdue","Perdue","Ossoff","Ossoff", "Ossoff","Perdue","Ossoff"),
  content = c(Perdue1, Perdue2, Perdue3, Ossoff1, Ossoff2, Ossoff3, Ossoff4))
candicate <- "healthcare voter tax help jobs"

#healthcare*voter*tax*help*jobs
sprintf(paste("Posterior probability for the email being from Perdue:", (1/15)*(1/15)*(1/15)*(1/15)*(1/
```

```
## [1] "Posterior probability for the email being from Perdue: 1.31687242798354e-06"
```

```r
sprintf(paste("Posterior probability for the email being from Ossoff:", (1/21)*(1/21)*(0/21)*(4/21)*(1/
```

```
## [1] "Posterior probability for the email being from Ossoff: 0"
```

Based on these results, I would predict that Perdue sent the mystery email. However, I wouldn't trust this conclusion to a great extent because the 0 posterior probability for Ossoff was a result of the zero count for the word `tax`. If the word `tax` was present in the Ossoff text, then the results would be flipped.

1(b)

```r
sprintf(paste("Posterior probability for the email being from Perdue:", (2/16)*(2/16)*(2/16)*(2/16)*(2/
```

```
## [1] "Posterior probability for the email being from Perdue: 3.0517578125e-05"
```

```r
sprintf(paste("Posterior probability for the email being from Ossoff:", (2/22)*(2/22)*(1/22)*(5/22)*(2/
```

```
## [1] "Posterior probability for the email being from Ossoff: 7.76151653823944e-06"
```

Based on these results, I would predict that Ossoff sent the mystery email. Smoothing makes sense because it tackles the problem of zero probabilities, and in case of a small or unrepresentative sample, smoothing helps the predictor generalize better to unseen data.

Part 3

2(a)

```
data <- data.table(read.csv('tripadvisor.csv'))

print(median(data$stars))
```

```
## [1] 4
```

```
#prop.table(table(data$stars))
data[stars > median(data$stars), true_class := '1']
data[stars <= median(data$stars), true_class := '0'] #put median in the negative class for a more balan

prop.table(table(data$true_class))
```

```
##
##         0         1
## 0.5581475 0.4418525
```

The median star rating is 4, and by assigning the median rating into the "negative" class, we arrive at a 44.185% positive - 55.815% negative proportion.

2(b)

```
data[stars == 5, anchor := 'positive']
data[stars < 5 & stars > 2, anchor := 'neutral']
data[stars <= 2, anchor := 'negative']
prop.table(table(data$anchor))
```

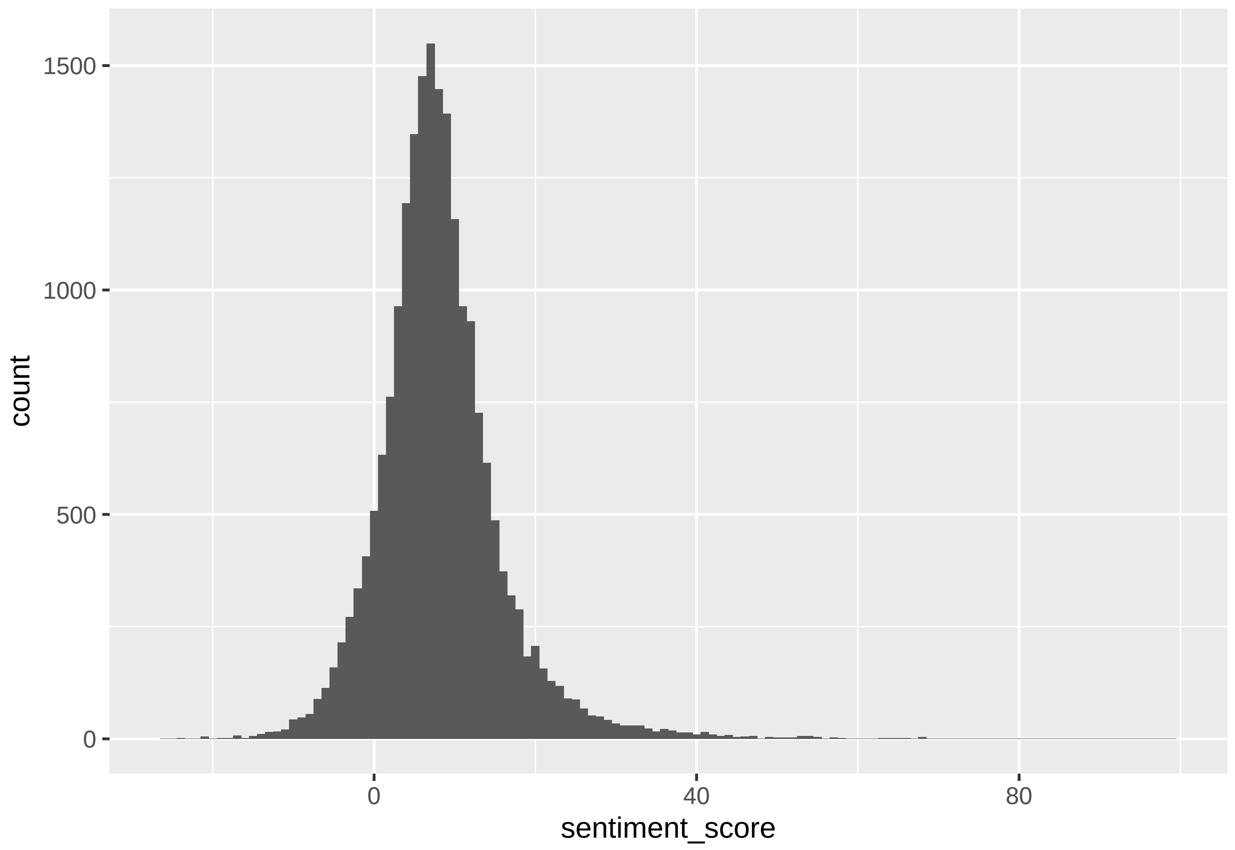
```
##
##  negative   neutral  positive
## 0.1568493 0.4012981 0.4418525
```

44.185% of the reviews are anchor positive, 40.129% neutral, and 15.685% negative.

3(a)

```
pos_dict <- c(read.delim('positive-words.txt', header = F)$V1)
neg_dict <- c(read.delim('negative-words.txt', header = F)$V1)

# replace apostrophes
data$text <- gsub(pattern = "'", "", data$text)

# count positive score
pos_dfm_3a <- as.data.frame(dfm(data$text, select = pos_dict, remove_punct = TRUE, tolower= TRUE))
data$pos_score <- rowSums(pos_dfm_3a[,c(-1)])
```

```r
# count negative score
neg_dfm_3a <- as.data.frame(dfm(data$text, select = neg_dict, remove_punct = TRUE, tolower= TRUE))
data$neg_score <- rowSums(neg_dfm_3a[,c(-1)])

# calculate sentiment score
data[, sentiment_score := pos_score - neg_score]

# get rid of columns
data[, pos_score := NULL]
data[, neg_score := NULL]

# histogram
ggplot(data, aes(x=sentiment_score)) + geom_histogram(binwidth=1)
```



For this question, the preprocessing steps that I took include replacing apostrophes, removing punctuation, and converting to lower case. After taking a look at the raw texts and both dictionaries, it seems to me that the tokens in the dictionaries were not stemmed, already lowered, and does not contain punctuation, and hence the preprocessing steps.

3(b)

```r
data[sentiment_score > 0, flag := "positive"]
data[sentiment_score <= 0, flag := "negative"]
#data[true_class == '1' & sentiment_score > 0, flag_consistent := 1]
#data[true_class == '0' & sentiment_score <= 0, flag_consistent := 1]
#data[is.na(flag_consistent), flag_consistent := 0]
```

```
# percent of reviews in each category
prop.table(table(data$flag))
```

```
##
##  negative  positive
## 0.1139037 0.8860963
```

```
#prop.table(table(data[true_class == '0']$flag_consistent))
```

88.609% of the reviews are labeled positive and 11.3904% negative. It seems like more reviews display positive sentiment than negative ones, although this might be due to the presence of more complex semantic structures such as irony and double negation that make it difficult to classify negative reviews correctly.

3(c)

```
#               pos true    neg true
# pos score
# neg score

conf_mat <- rbind(prop.table(table(data[flag == "positive"]$true_class))[c(2,1)],
                  prop.table(table(data[flag == "negative"]$true_class))[c(2,1)])

# confusion matrix
conf_mat
```

```
##              1         0
## [1,] 0.4931431 0.5068569
## [2,] 0.0428449 0.9571551
```

```
# accuracy = (TP + TN) / (TP + FP + TN + FN)
sprintf(paste("Accuracy:", sum(diag(conf_mat))/sum(conf_mat)))
```

```
## [1] "Accuracy: 0.725149119464951"
```

```
# recall = TP / (TP + FN)
sprintf(paste("Recall:", conf_mat[2,2]/sum(conf_mat[2,])))
```

```
## [1] "Recall: 0.957155098543273"
```

```
# precision = TP / (TP + FP)
sprintf(paste("Precision:", conf_mat[2,2]/sum(conf_mat[,2])))
```

```
## [1] "Precision: 0.653789125977112"
```

```
# f1 score
sprintf(paste("F-1 Score:", 2*(conf_mat[2,2]/sum(conf_mat[2,])*conf_mat[2,2]/sum(conf_mat[,2]))/(conf_ma
```

```
## [1] "F-1 Score: 0.776907835511749"
```

```
# baseline accuracy
sprintf(paste("Baseline Accuracy:", max(prop.table(table(data$true_class)))))
```

```
## [1] "Baseline Accuracy: 0.558147479381192"
```

This classifier achieved an accuracy of 0.725, higher than the baseline accuracy which is 0.448, and thus it is an improvement in performance.

4(a)

The preprocessing steps I took, on top of replacing apostrophes and removing punctuation, include stemming and removing stop words. This is because we are no longer comparing the texts to a dictionary, and stemming and removing stop words help with the accuracy.

4(b)

```r
# split sample into training & test sets
prop_train <- 0.8
ids <- 1:nrow(data)

ids_train <- sample(ids, ceiling(prop_train*length(ids)), replace = FALSE)
ids_test <- ids[-ids_train]
train_set <- data[ids_train,]
test_set <- data[ids_test,]

# get dfm for each set
train_dfm <- dfm(train_set$text, stem = TRUE, remove_punct = TRUE, remove = stopwords("english"))
test_dfm <- dfm(test_set$text, stem = TRUE, remove_punct = TRUE, remove = stopwords("english"))

# match test set dfm to train set dfm features
test_dfm <- dfm_match(test_dfm, features = featnames(train_dfm))

# train model on the training set using Laplace smoothing
nb_model_sm <- textmodel_nb(train_dfm, train_set$true_class, smooth = 1, prior = "uniform")

# evaluate on test set
predicted_class_sm <- predict(nb_model_sm, newdata = test_dfm, force=TRUE)

# baseline
baseline_acc <- max(prop.table(table(test_set$true_class)))

# get confusion matrix
cmat_sm <- table(test_set$true_class, predicted_class_sm)
nb_acc_sm <- sum(diag(cmat_sm))/sum(cmat_sm) # accuracy = (TP + TN) / (TP + FP + TN + FN)
nb_recall_sm <- cmat_sm[2,2]/sum(cmat_sm[2,]) # recall = TP / (TP + FN)
nb_precision_sm <- cmat_sm[2,2]/sum(cmat_sm[,2]) # precision = TP / (TP + FP)
nb_f1_sm <- 2*(nb_recall_sm*nb_precision_sm)/(nb_recall_sm + nb_precision_sm)

# print
cat(
  "Baseline Accuracy: ", baseline_acc, "\n",
  "Accuracy:",  nb_acc_sm, "\n",
  "Recall:",  nb_recall_sm, "\n",
  "Precision:",  nb_precision_sm, "\n",
  "F1-score:", nb_f1_sm
)
```

```
## Baseline Accuracy:  0.5610054
##  Accuracy: 0.7525622
##  Recall: 0.8243469
##  Precision: 0.6799633
##  F1-score: 0.7452261
```

```
print(cmat_sm)
```

```
##    predicted_class_sm
##        0    1
##   0 1601  698
##   1  316 1483
```

4(c)

```
# train model on the training set using Laplace smoothing and docfreq prior
nb_model_sm_docfreq <- textmodel_nb(train_dfm, train_set$true_class, smooth = 1, prior = "docfreq")

# evaluate on test set
predicted_class_sm_docfreq <- predict(nb_model_sm_docfreq, newdata = test_dfm, force=TRUE)

# get confusion matrix
cmat_sm_docfreq <- table(test_set$true_class, predicted_class_sm_docfreq)
nb_acc_sm_docfreq <- sum(diag(cmat_sm_docfreq))/sum(cmat_sm_docfreq) # accuracy = (TP + TN) / (TP + FP
nb_recall_sm_docfreq <- cmat_sm_docfreq[2,2]/sum(cmat_sm_docfreq[2,]) # recall = TP / (TP + FN)
nb_precision_sm_docfreq <- cmat_sm_docfreq[2,2]/sum(cmat_sm_docfreq[,2]) # precision = TP / (TP + FP)
nb_f1_sm_docfreq <- 2*(nb_recall_sm*nb_precision_sm)/(nb_recall_sm + nb_precision_sm)

# print
cat(
  "Baseline Accuracy: ", baseline_acc, "\n",
  "Accuracy:",  nb_acc_sm_docfreq, "\n",
  "Recall:",  nb_recall_sm_docfreq, "\n",
  "Precision:",  nb_precision_sm_docfreq, "\n",
  "F1-score:", nb_f1_sm_docfreq
)
```

```
## Baseline Accuracy:  0.5610054
##  Accuracy: 0.7530503
##  Recall: 0.8160089
##  Precision: 0.6831084
##  F1-score: 0.7452261
```

```
print(cmat_sm_docfreq)
```

```
##    predicted_class_sm_docfreq
##        0    1
##   0 1618  681
##   1  331 1468
```

I expected this to change the performance of the Naive Bayes prediction, and it in fact did. This is because with the `docfreq` prior, the class priors will be taken from the relative proportions of the class documents used in the training set, but there may be nothing informative in the relative numbers of documents used to train a classifier other than the relative availability of the documents. The accuracy is a slight improvement from the previous predictor, since it is more catered to this specific sample.

4(d)

```r
# train model on the training set without smoothing
nb_model <- textmodel_nb(train_dfm, train_set$true_class, smooth = 0, prior = "uniform")

# evaluate on test set
predicted_class <- predict(nb_model, newdata = test_dfm)

# get confusion matrix (see slide 35 from class)
cmat <- table(test_set$true_class, predicted_class)
cmat
```

```
##    predicted_class
##        0    1
##   0 1641  658
##   1  769 1030
```

```r
nb_acc <- sum(diag(cmat))/sum(cmat) # accuracy = (TP + TN) / (TP + FP + TN + FN)
nb_recall <- cmat[2,2]/sum(cmat[2,]) # recall = TP / (TP + FN)
nb_precision <- cmat[2,2]/sum(cmat[,2]) # precision = TP / (TP + FP)
nb_f1 <- 2*(nb_recall*nb_precision)/(nb_recall + nb_precision)

# print
cat(
  "Baseline Accuracy: ", baseline_acc, "\n",
  "Accuracy:",  nb_acc, "\n",
  "Recall:",  nb_recall, "\n",
  "Precision:",  nb_precision, "\n",
  "F1-score:", nb_f1
)
```

```
## Baseline Accuracy:  0.5610054
##  Accuracy: 0.6517814
##  Recall: 0.5725403
##  Precision: 0.6101896
##  F1-score: 0.5907657
```

The accuracy is lower than those of the previous two models. This is because without smoothing there are many zero probabilities in the Naive Bayes model, and so it does not generalize well to unseen data.

4(e)

Punctuation and Capitalization may also help with sentiment classification.

5(a)

```r
filenames <- list.files(path = "manifestos")

# Party name and year are in the filename -- we can use regex to extract these to use as our docvars
#party <- unlist(regmatches(unlist(filenames), gregexpr("^[[:alpha:]]{3}", unlist(filenames))))
party <- c("Con", "Con", "Lab", "Lab")
year <- unlist(regmatches(unlist(filenames), gregexpr("[[:digit:]]+", unlist(filenames))))
```

```
# Make a corpus with docvars from this data
cons_labour_manifestos <- corpus(readtext("manifestos/*.txt"))
docvars(cons_labour_manifestos, field = c("party", "year")) <- data.frame(cbind(party, year))

# But we're going to use a dataframe
cons_labour_df <- data.table(text = as.character(cons_labour_manifestos),
                         class = party,
                         year = as.integer(year))

anchor_labor <- cons_labour_df[class == "Lab" & year == 1945]
anchor_cons <- cons_labour_df[class == 'Con'& year == 1983]

train_set <- rbind(anchor_labor, anchor_cons)

train_dfm <- dfm(train_set$text, remove_punct = TRUE, remove = stopwords("english"))

# apply smoothing
ws_base <- textmodel_wordscores(train_dfm,
                                y = (2 * as.numeric(train_set$class == "Lab"))-1,
                                smooth=1)

# top 10 highest
head(sort(ws_base$wordscores, decreasing = TRUE), 10)
```

```
##     parties    standard         war         won  production    ministry
##   0.8904316   0.8648900   0.8608349   0.8470645   0.8470645   0.8470645
##         let     victory   factories progressive
##   0.8238206   0.8238206   0.8238206   0.8238206
```

```
# top 10 lowest
head(sort(ws_base$wordscores), 10)
```

```
##   continue  encourage        now      shall   training       four          £
## -0.8708833 -0.8481568 -0.8335039 -0.8176684 -0.8157209 -0.8053246 -0.8053246
##     scheme        buy      firms
## -0.8053246 -0.7936850 -0.7936850
```

I chose to use smoothing for better generalization to the text set.

5(b)

```
test_labor <- cons_labour_df[class == 'Lab' & year != 1945]
test_cons <- cons_labour_df[class == 'Con' & year != 1983]
test_set <- rbind(test_labor, test_cons)

test_dfm <- dfm(test_set$text, remove_punct = TRUE, remove = stopwords("english"))
```

```
## Warning: 'dfm.character()' is deprecated. Use 'tokens()' first.

## Warning: '...' should not be used for tokens() arguments; use 'tokens()' first.

## Warning: 'remove' is deprecated; use dfm_remove() instead
```

```
predict(ws_base, newdata = test_dfm, rescaling = "none", level = 0.95)
```

```
## Warning: 849 features in newdata not used in prediction.
##       text1       text2
## -0.01034205 -0.12814230
```

I was able to correctly predicted the Conservative text but not the Labor text. The warning message suggests that 849 features in the new data are not used in prediction. This could be problematic as it suggests that the model might be highly dependent on a relatively small selection of features and its preditive power might be limited. The shrinkage of the score is high.

5(c)

```
predict(ws_base, newdata = test_dfm, rescaling = "lbg", level = 0.95)
```

```
##      text1      text2
##  0.9307578 -1.0692422
```

The predictions are correct with the standard LBG rescaling.

6(a)

```
q6data <- data[0:1000,]
q6data$text <- gsub(pattern = "'", "", q6data$text)

q6dfm <- dfm(q6data$text, stem = TRUE, remove_punct = TRUE, remove = stopwords("english")) %>% convert(
```

The preprocessing steps I take include removing stop words, punctuation, apostrophes, and stemming. This is good for this task because SVM is computationally expensive, and limiting the number of features can alleviate this problem. An advantage offered by SVM is that it has a larger hypothesis space, and if faces fewer constraints compared to the dictionary approach because we don't assume domain knowledge in building a dictionary beforehand.

6(b)

```
ids_train <- createDataPartition(1:nrow(q6dfm), p = 0.1, list = FALSE, times = 1)

train_x <- q6dfm[ids_train, ] %>% as.data.frame() # train set data
train_y <- q6data$true_class[ids_train] %>% as.factor() # train set labels
test_x <- q6dfm[-ids_train, ] %>% as.data.frame() # test set data
test_y <- q6data$true_class[-ids_train] %>% as.factor() # test set labels

# baseline
baseline_acc <- max(prop.table(table(test_y)))

# define training options
trctrl <- trainControl(method = "none")

# now we will have train / test / validation
val_x <- test_x
val_y <- test_y
trctrl <- trainControl(method = "cv", number = 5)

# svm - linear
svm_mod_linear <- train(x = train_x, y = train_y, method = "svmLinear", trControl = trctrl)
```

```r
# predict on heldout validation data
svm_linear_pred <- predict(svm_mod_linear, newdata = val_x)
q6cmat <- table(val_y, svm_linear_pred)
sprintf(paste("Highest Accuracy:", sum(diag(q6cmat))/sum(q6cmat)))
```

```
## [1] "Highest Accuracy: 0.697777777777778"
```

```r
#results <- data.frame(train_size = NA, accuracy = NA)
#for (i in 1:9) {
#  ids_train <- createDataPartition(1:nrow(q6dfm), p = 0.1*i, list = FALSE, times = 1)
#  train_x <- q6dfm[ids_train, ] %>% as.data.frame()
#  train_y <- data$true_class[ids_train] %>% as.factor()
#  val_x <- q6dfm[-ids_train, ] %>% as.data.frame()
#  val_y <- data$true_class[-ids_train] %>% as.factor()
#  trctrl <- trainControl(method = "cv", number = 5)
#  svm_mod_linear <- train(x = train_x, y = train_y, method = "svmLinear", trControl = trctrl)
#  svm_linear_pred <- predict(svm_mod_linear, newdata = val_x)
#  svm_linear_cmat <- confusionMatrix(svm_linear_pred, val_y)
#  results[i,] <- c(0.1*i, svm_linear_cmat$overall[1])
#}
```

In terms of accuracy, the model performs relatively well, especially considering that the baseline accuracy is 66.44%.

6(c)

```r
q6cdata <- data[1:100,]
q6cdata$text <- gsub(pattern = "'", "", q6cdata$text)
q6cdfm <- dfm(q6cdata$text, stem = TRUE, remove_punct = TRUE, remove = stopwords("english")) %>% conver

ids_train <- createDataPartition(1:nrow(q6cdfm), p = 0.8, list = FALSE, times = 1)
train_x <- q6cdfm[ids_train, ] %>% as.data.frame()
train_y <- q6cdata$true_class[ids_train] %>% as.factor()
val_x <- q6cdfm[-ids_train, ] %>% as.data.frame()
val_y <- q6cdata$true_class[-ids_train] %>% as.factor()
trctrl <- trainControl(method = "cv", number = 5)
svm_mod_logit <- train(x = train_x, y = train_y, method = "glm", trControl = trctrl, family = 'binomial
svm_logit_pred <- predict(svm_mod_logit, newdata = val_x)
svm_logit_cmat <- table(svm_logit_pred, val_y)
sprintf(paste("Accuracy:", sum(diag(svm_logit_cmat))/sum(svm_logit_cmat)))
```

```
## [1] "Accuracy: 0.7"
```

The accuracy achieved by the logistic model is slightly higher than the baseline accuracy, which means that the model is an improvement. One way to help the algorithm converge is to lower the learning rate, but that would induce longer runtime. It is also lower than the SVM accuracy.

7(a)

```r
q7data <- data[0:500,]
q7data$text <- gsub(pattern = "'", "", q7data$text)
```

```r
q7dfm <- dfm(q7data$text, stem = TRUE, remove_punct = TRUE, remove = stopwords("english")) %>% convert(

ids_train <- createDataPartition(1:nrow(q7dfm), p = 0.8, list = FALSE, times = 1)

train_x <- q7dfm[ids_train, ] %>% as.data.frame() # train set data
train_y <- q7data$true_class[ids_train] %>% as.factor() # train set labels
test_x <- q7dfm[-ids_train, ] %>% as.data.frame() # test set data
test_y <- q7data$true_class[-ids_train] %>% as.factor() # test set labels
```

7(b)

```r
rf.base <- randomForest(x = train_x, y = train_y, importance = TRUE)
token_importance <- round(importance(rf.base, 2), 2)
head(rownames(token_importance)[order(-token_importance)],10)
```

```
##  [1] "great"     "love"      "excel"     "staff"     "night"     "room"
##  [7] "fantast"   "moment"    "wonder"    "recommend"
```

7(c)

```r
predict_test <- predict(rf.base, newdata = test_x)
cmat <- table(test_y, predict_test)

nb_acc <- sum(diag(cmat))/sum(cmat) # accuracy = (TP + TN) / (TP + FP + TN + FN)
nb_recall <- cmat[2,2]/sum(cmat[2,]) # recall = TP / (TP + FN)
nb_precision <- cmat[2,2]/sum(cmat[,2]) # precision = TP / (TP + FP)
nb_f1 <- 2*(nb_recall*nb_precision)/(nb_recall + nb_precision)

cat(
"Accuracy:", nb_acc, "\n",
"Recall:", nb_recall, "\n",
"Precision:", nb_precision, "\n",
"F1-score:", nb_f1
)
```

```
## Accuracy: 0.72
##  Recall: 0.03448276
##  Precision: 1
##  F1-score: 0.06666667
```

7(d)

```r
mtry <- sqrt(ncol(train_x))

# 0.5X
rf.base_0.5 <- randomForest(x = train_x, y = train_y, mtry = 0.5*mtry, importance = TRUE)
predict_test_0.5 <- predict(rf.base_0.5, newdata = test_x)
cmat_0.5 <- table(test_y, predict_test_0.5)
sum(diag(cmat_0.5))/sum(cmat_0.5)
```

```
## [1] 0.71
```

```r
# 1.5X
rf.base_1.5 <- randomForest(x = train_x, y = train_y, mtry = 1.5*mtry, importance = TRUE)
predict_test_1.5 <- predict(rf.base_1.5, newdata = test_x)
cmat_1.5 <- table(test_y, predict_test_1.5)
sum(diag(cmat_1.5))/sum(cmat_1.5)
```

```
## [1] 0.72
```

`mtry` = 1.5*sqrt(# of features) yielded the slightly higher accuracy.