Homework 2 DS-GA 1015

Alexandre Vives, N14948495

4/11/2022

Please list everyone you collaborated with on this assignment

---

```
library(quanteda)
library(quanteda.textmodels)
library(caret)
library(readtext)
library(tidyverse)
library(randomForest)
```

Optional Section

Our area of research is a combination of statistics and machine learning in textual data. We are trying to define the sentiment of words used in online reviews. This is very interesting, because by creating a sentiment dictionary with words used in online reviews, we will then be able to infer the rating and the sentiment of all the available product reviews online.

Our main question is if we can utilize the vast pool of amazon reviews that are available, to infer the sentiment of each word.

Our independent variables are the review's text, author and product.

Our outcome variable is the score assigned to the review, which ranges from 1 to 5 stars.

Our text comes from an amazon reviews dataset in Kaggle. We already have the data and it is clean enough to start using it. Moreover, we will choose the most relevant features (apart from the text itself).

We will use SVM and logistic regression classification methods to answer that question. The reason we are choosing these models is because they are very efficient when it comes to a classification task. They will enable us to design a good word dictionary tailored to product reviews which can then be used to even rate a review again (this could be useful because people assign stars to products depending on personal standards).

Partner: Ilias Arvanitakis

Question 1

a) Perform Naive Bayes by hand.

Mystery email: "healthcare voter tax help jobs"

P("healthcare"|Perdue) = 1/15 P("voter"|Perdue) = 1/15 P("tax"|Perdue) = 1/15 P("help"|Perdue) = 1/15 P("jobs"|Perdue) = 1/15

P("healthcare"|Ossoff) = 1/21 P("voter"|Ossoff) = 1/21 P("tax"|Ossoff) = 0/21 P("help"|Ossoff) = 4/21 P("jobs"|Ossoff) = 1/21

P(Perdue|Mystery email) = k P(Mystery email|Perdue)P(Perdue) = $\frac{1}{15}\frac{1}{15}\frac{1}{15}\frac{1}{15}\frac{1}{15}\frac{3}{7} = 5.64 \times 10^{-7}$

P(Ossoff|Mystery email) = k P(Mystery email|Ossoff)P(Ossoff) = $\frac{1}{21}\frac{1}{21}\frac{0}{21}\frac{4}{21}\frac{1}{21}\frac{4}{7} = 0$

Naive Bayes has two main disadvantages: It assumes independence between tokens (that is what gives it the name "Naive") and if a test token does not appear in the training set, it gets assigned a probability of O.

The second disadvantage is the main reason why I do not trust this result, the probability of seeing the input "healthcare voter tax helps jobs" given the email was from Ossoff ends up being O just because the word "tax" was not on the Ossoff training set.

We predict Perdue.

b) Perform Naive Bayes by hand using smoothing.

Mystery email: "healthcare voter tax help jobs"

P("healthcare"|Perdue) = 1/15 -> 2/43 P("voter"|Perdue) = 1/15 -> 2/43 P("tax"|Perdue) = 1/ -> 2/43 P("help"|Perdue) = 1/15 -> 2/43 P("jobs"|Perdue) = 2/15 -> 2/43

P("healthcare"|Ossoff) = 1/21 -> 2/49 P("voter"|Ossoff) = 1/21 -> 2/49 P("tax"|Ossoff) = 0/21 -> 1/49 P("help"|Ossoff) = 4/21 -> 5/49 P("jobs"|Ossoff) = 1/21 -> 2/49

P(Perdue|Mystery email) = k P(Mystery email|Perdue)P(Perdue) = $\frac{2}{43}\frac{2}{43}\frac{2}{43}\frac{2}{43}\frac{2}{43}\frac{3}{7} = 9.32 \times 10^{-8}$

P(Ossoff|Mystery email) = k P(Mystery email|Ossoff)P(Ossoff) = $\frac{2}{49}\frac{2}{49}\frac{1}{49}\frac{5}{49}\frac{2}{49}\frac{4}{7} = 8.09 \times 10^{-8}$

Smoothing can help in scenarios where there is a token in test data that did not appear in train data because after smoothing, the probability will no longer be 0. Additionally, as mentioned in the question, this is just a sample of each candidate's language, smoothing will allow us to apply a probability to each potential word they might use in the future.

We predict Perdue.

QUESTION 2

a) Divide the reviews using the median.

```
raw_df <- read.csv(file = 'tripadvisor.csv') # Read the data
# Assign the true labels
raw_df$label[raw_df$stars >= median(raw_df$stars)] <- "positive"
raw_df$label[raw_df$stars < median(raw_df$stars)] <- "negative"

df <- data.frame(raw_df) # Make a copy of the raw data

sprintf('The median star rating is %d stars', median(df$stars))
```

```
## [1] "The median star rating is 4 stars"
```

```
sprintf('Proportion of positive labels: %f', sum(df$label == 'positive') / nrow(df))
```

```
## [1] "Proportion of positive labels: 0.736567"
```

```
sprintf('Proportion of negative labels: %f', sum(df$label == 'negative') / nrow(df))
```

```
## [1] "Proportion of negative labels: 0.263433"
```

b) Create the anchor column

```
df$anchor[df$stars == 5] <- "positive"
df$anchor[df$stars %in% c(3,4)] <- "neutral"
df$anchor[df$stars %in% c(1,2)] <- "negative"

sprintf('Proportion of positive labels: %f', sum(df$anchor == 'positive') / nrow(df))
```

```
## [1] "Proportion of positive labels: 0.441853"
```

```
sprintf('Proportion of negative labels: %f', sum(df$anchor == 'neutral') / nrow(df))
```

```
## [1] "Proportion of negative labels: 0.401298"
```

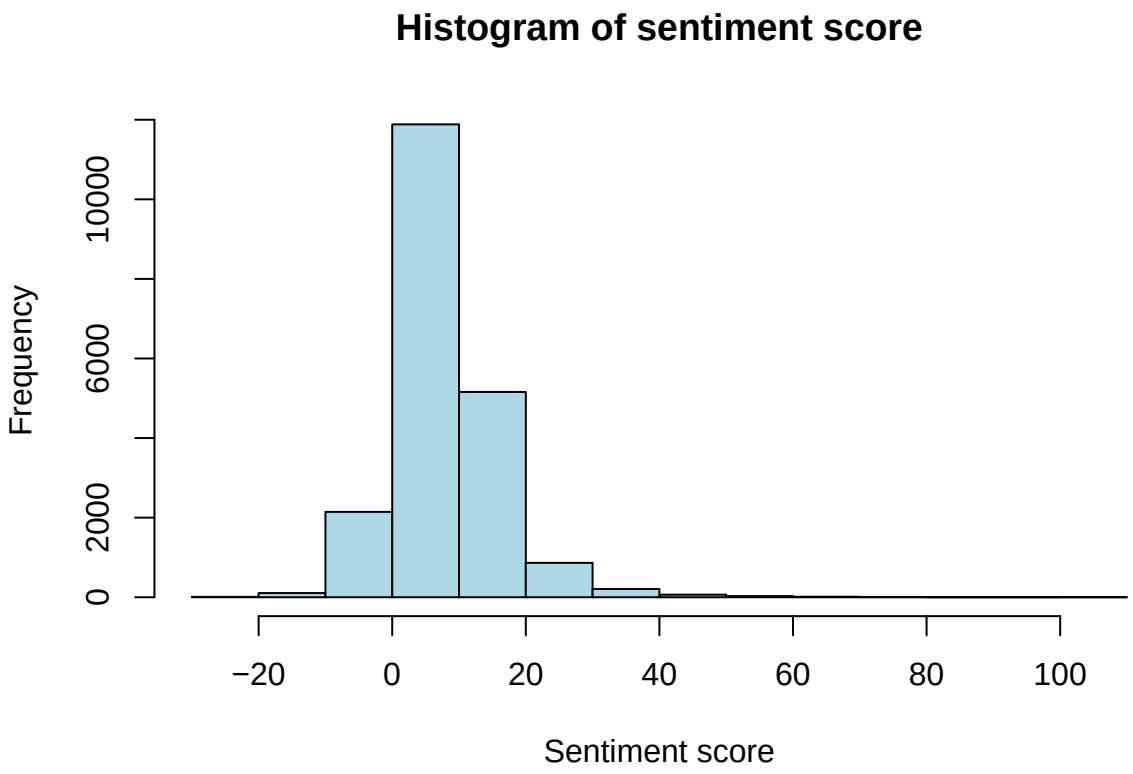
```
sprintf('Proportion of negative labels: %f', sum(df$anchor == 'negative') / nrow(df))
```

```
## [1] "Proportion of negative labels: 0.156849"
```

QUESTION 3

a) Explain appropriate pre-processing steps and create sentiment score.

```
df$text <- gsub('[[:punct:]]+', '', df$text) #Removes punctuation
df$text <- gsub('[[:digit:]]+', '', df$text) #Removes numbers

positive_words <- read.csv(file = 'positive-words.txt', header=FALSE)
negative_words <- read.csv(file = 'negative-words.txt', header=FALSE)

for (idx in 1:nrow(df)) {
  review <- df$text[idx]
  num_pos <- 0
  num_neg <- 0

  for (word in strsplit(review, ' ')[[1]]) {
    if (word %in% positive_words$V1)
      num_pos <- num_pos + 1
    else if (word %in% negative_words$V1)
      num_neg <- num_neg + 1
  }

  df$sentiment_score[idx] = num_pos - num_neg
}

hist(df$sentiment_score, main="Histogram of sentiment score", xlab="Sentiment score",
     col="lightblue")
```



In this exercise there is no need for many pre-processing because we will only use the words that match any postive/negative words in our lists. The only benefit of pre-processing would be to shorten the reviews and thus reduce the time it takes to loop through them in the future, for that reason I will simply remove punctuation and numbers (because I know those won't be in the positive/negative words list).

b) Create a vector of dichotomous variables and discuss the results.

```
sentiment_vector <- vector()
for (idx in 1:nrow(df)) {
  if (df$sentiment_score[idx] > 0)
    sentiment_vector[idx] <- "positive"
  else
    sentiment_vector[idx] <- "negative"
}

sprintf('There are %.2f%% of positive sentiment reviews',
        100*sum(sentiment_vector == "positive") / length(sentiment_vector))
```

```
## [1] "There are 88.96% of positive sentiment reviews"
```

```
sprintf('There are %.2f%% of negative sentiment reviews',
        100*sum(sentiment_vector == "negative") / length(sentiment_vector))
```

```
## [1] "There are 11.04% of negative sentiment reviews"
```

We can see that the positive reviews are twice the negative ones, making it an imbalanced data set. This will influence the way we evaluate the model.

c) Compute a confusion matrix and use it to calculate accuracy, precision, recall and F1 score.

```
accuracy_metrics <- function(predicted_values, true_values) {

  TP = 0
  TN = 0
  FP = 0
  FN = 0

  for (idx in 1:length(predicted_values)) {
    if (predicted_values[idx] == "positive" && true_values[idx] == "positive" ) TP <- TP + 1
    else if (predicted_values[idx] == "negative" && true_values[idx] == "negative" ) TN <- TN + 1
    else if (predicted_values[idx] == "positive" && true_values[idx] == "negative" ) FP <- FP + 1
    else if (predicted_values[idx] == "negative" && true_values[idx] == "positive" ) FN <- FN + 1
  }

  Accuracy = (TP + TN)/(TP + TN + FP + FN)
  Precision = TP/(TP + FP)
  Recall = TP/(TP + FN)
  Specificity = TN/(TN + FP)
  F1_score = 2*Precision*Recall/(Precision + Recall)

  cat(sprintf("Accuracy: %.2f%%\n", 100*Accuracy))
  cat(sprintf("Precision: %.2f%%\n", 100*Precision))
  cat(sprintf("Recall: %.2f%%\n", 100*Recall))
  cat(sprintf("Specificity: %.2f%%\n", 100*Specificity))
  cat(sprintf("F1-Score: %.2f%%", F1_score))
}

accuracy_metrics(sentiment_vector, df$label)
```

```
## Accuracy: 82.30%
## Precision: 81.45%
## Recall: 98.38%
## Specificity: 37.37%
## F1-Score: 0.89%
```

```
#                         PRED
#                   positive   negative
#                +----------------------+
#      positive  | TP=14,848 | FN=245    |
#TRUE            +-----------------------+
#      negative  | FP=3,381  | TN=2,017 |
#                +----------------------+
```

How would you evaluate the performance of this classifier?

The performance of this classifier should be evaluated depending on how balanced the target variable is. We saw that there are 73% positive classes and 26% negative classes, therefore it is unbalanced. In this situations, we should not use the accuracy measure (because predicting all positive we would get 73%) therefore we should use F1-score (which takes the harmonic mean between precision and recall).

QUESTION 4

a) Explain any pre-processing needed for this task.

When training a Naive Bayes model it is useful to reduce the amount of tokens to avoid assigning 0 probabilities to tokens that happen to not be included in the training set, therefore stemming and eliminating capitalized letters (lowering) is very useful. Additionally, I will remove punctuation because it just does not provide value.

b) Create a Naive Bayes model and compute all confusion matrix values.

```
train_test_split = 0.2*nrow(df)

train_dfm <- dfm(raw_df$text[1:train_test_split], stem=TRUE, remove_punct=TRUE,
                 tolower=TRUE, remove = stopwords("english"))
train_y <- raw_df$label[1:train_test_split]
test_dfm <- dfm(raw_df$text[-(1:train_test_split)], stem=TRUE, remove_punct=TRUE,
                tolower=TRUE, remove = stopwords("english"))
test_y <- raw_df$label[-(1:train_test_split)]

test_dfm <- dfm_match(test_dfm, features=featnames(train_dfm))

nb_model <- textmodel_nb(x=train_dfm, y=train_y, smooth=1, prior='uniform')

predicted_class <- predict(nb_model, newdata=test_dfm)

accuracy_metrics(predicted_class, test_y)
```

```
## Accuracy: 88.24%
## Precision: 88.76%
## Recall: 96.51%
## Specificity: 63.61%
## F1-Score: 0.92%
```

```
#                         PRED
#                   positive   negative
#                +----------------------+
#      positive  | TP=11,843 | FN=428    |
#TRUE            +----------------------+
#      negative  | FP=1,500  | TN=2,622 |
#                +----------------------+
```

c)

```
nb_model2 <- textmodel_nb(x=train_dfm, y=train_y, smooth=2, prior='docfreq')

predicted_class <- predict(nb_model2, newdata=test_dfm)

accuracy_metrics(predicted_class, test_y)
```

```
## Accuracy: 86.16%
```

```
## Precision: 85.13%
## Recall: 98.78%
## Specificity: 48.62%
## F1-Score: 0.91%
```

```
#                        PRED
#                  positive   negative
#               +----------------------+
#      positive | TP=12,121 | FN=150   |
#TRUE           +----------------------+
#      negative | FP=2,118  | TN=2,004 |
#               +----------------------+
```

Would you expect this to change the performance of Naive Bayes predictions? Why?

It should. A document frequency prior would be equal to a uniform prior if the classes were balanced, but in this case they are unbalanced (73% positives and 27% negatives). Thus, using this prior would make the model be more inclined to predict positive rather than negative (at least before being updated by a lot of observations). This can be observed on the confusion matrix, we can see that the True Positives increased but at the expense of some True Negatives, but overall, the model is increased the F1-score by 2%.

How would you evaluate the performance of this classifier?

The performance of this classifier should be evaluated depending on how balanced the target variable is. We saw that there are 73% positive classes and 26% negative classes, therefore it is unbalanced. In this situations, we should not use the accuracy measure (because predicting all positive we would get 73%) therefore we should use F1-score (which takes the harmonic mean between precision and recall).

d)

```
nb_model3 <- textmodel_nb(x=train_dfm, y=train_y, smooth=0, prior='uniform')

predicted_class <- predict(nb_model3, newdata=test_dfm)

accuracy_metrics(predicted_class, test_y)
```

```
## Accuracy: 65.53%
## Precision: 84.38%
## Recall: 66.21%
## Specificity: 63.51%
## F1-Score: 0.74%
```

```
#                        PRED
#                  positive   negative
#               +----------------------+
#      positive | TP=8,125  | FN=4,146 |
#TRUE           +----------------------+
#      negative | FP=1,504  | TN=2,618 |
#               +----------------------+
```

How does the accuracy compare to the previous models? Why might this be?

The accuracy for the un-smoothed model is much worse (50% accuracy with 5000 false negatives). This is due to the removal of the smooth parameter, which is very important for Naive Bayes because it avoids the problem of having a feature in the testing that was not assigned a probability during training.

e)

The date the review was posted, the author of the review, the hotel reviewed, the location of the hotel and the age of the author.

QUESTION 5

a)

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

party <- unlist(regmatches(unlist(filenames), gregexpr("^[[:alpha:]]{3}", unlist(filenames))))
year <- unlist(regmatches(unlist(filenames), gregexpr("[[:digit:]]+", unlist(filenames))))

cons_labour_manifestos <- corpus(readtext("manifestos/*.txt"))
docvars(cons_labour_manifestos, field = c("party", "year") ) <- data.frame(cbind(party, year))

cons_labour_df <- tibble(text = texts(cons_labour_manifestos), class = party,
                         year = as.integer(year))

training_idx <- which(cons_labour_df$year %in% c(1945, 1983))
testing_idx <- which(cons_labour_df$year %in% c(1951, 1979))

train_dfm <- dfm(cons_labour_df$text[training_idx], stem=TRUE, remove_punct=TRUE,
                 tolower=TRUE, drop_number=TRUE, remove = stopwords("english"))
train_y <- 2 * as.numeric(cons_labour_df$class[training_idx] == "Lab") - 1
test_dfm <- dfm(cons_labour_df$text[testing_idx], stem=TRUE, remove_punct=TRUE,
                tolower=TRUE, drop_number=TRUE, remove = stopwords("english"))
test_y <- 2 * as.numeric(cons_labour_df$class[testing_idx] == "Lab") - 1

model_ws <- textmodel_wordscores(train_dfm, y = train_y, smooth=1)

lab_features <- sort(model_ws$wordscores, decreasing = TRUE)
print(lab_features[2:11]) #First value is the year
```

```
##       won  ministri       war       let    declar   victori   section   suitabl
## 0.8563045 0.8563045 0.8482564 0.8343389 0.8343389 0.8343389 0.8343389 0.8343389
##   struggl   depress
## 0.8044463 0.8044463
```

```
con_features <- sort(model_ws$wordscores, decreasing = FALSE)
con_features[2:11] #First value is the year
```

```
##     scheme      train    pension        now    continu      shall    council
## -0.8674489 -0.8384386 -0.8310429 -0.8229377 -0.8108384 -0.8062034 -0.8041467
##       four      spend          â
## -0.7931718 -0.7931718 -0.7931718
```

I applied smoothing because when this model was ran without smoothing, there were many extreme values (assigned 1 and -1). Therefore in order to have a clearer hierarchy of wordscores, a smoothing of 1 was applied (the extremes are reduced and they regress to the mean, which in this case is 0).

b)

```
predict(model_ws, newdata = test_dfm)
```

```
## Con1979.txt Lab1951.txt
## -0.12766728 -0.02304218
```

Did you correctly predict the party?

The conservative party was correctly classified (with not much confidence though), but the labor party was incorrectly classified.

What does the warning message suggests?

The warning message suggests that we should match the features of the test set to the train set (or else its probability is 0).

What can you tell about the shrinkage of the scores? Can this be problematic? Why?

As mentioned in the document "Understanding Wordscores" by Will Lowe, the shrinkage effect is common in applications where the mean and variance of document scores can be expected to be approximately constant, like it is the case in manifestos.

It can be problematic as the predicted document scores are usually very close to the mean of the training document scores (which in our case is 0), and we can see that in our case our prediction are indeed very close to 0.

LBG rescaling fixes this issue by applying a formula that separates the predicted scores.

Reference: https://faculty.washington.edu/jwilker/559/Lowe.pdf

c)

```
predicted_cass <- predict(model_ws, newdata = test_dfm, rescaling='lbg')
```

How does your results compare to the previous specification without rescaling?

The results are much better, both parties are predicted correctly.

QUESTION 6

a)

The pre-processing steps are similar to the previous ones, I will remove punctuation (provides no value), stopwords (could overtake the dataset) and finally stemming and lowering to reduce the a amount of unique tokens.

Models such as SVM and Naive Bayes have the advantage over word dictionaries in terms of its flexibility. The coefficients assigned to each feature are deduced using the data itself, whereas the coefficients of a word dictionary are a generalization for each specific case (hotel reviews in this case).

b)

```
small_df <- raw_df[1:1000, ]

splits <- c(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9)
for (split in splits) {
  train_test_split <- split*nrow(small_df)

  train_dfm <- dfm(small_df$text[1:train_test_split], stem=TRUE, remove_punct=TRUE,
                   tolower=TRUE, remove = stopwords("english"))
  train_y <- small_df$label[1:train_test_split]

  test_dfm <- dfm(small_df$text[-(1:train_test_split)], stem=TRUE, remove_punct=TRUE,
                  tolower=TRUE, remove = stopwords("english"))
  test_y <- small_df$label[-(1:train_test_split)]

  test_dfm <- dfm_match(test_dfm, features=featnames(train_dfm))

  trainControl <- trainControl(method = "cv", number = 5)
  svm_model <- train(x=train_dfm, y=train_y, method="svmLinear", trControl=trainControl)

  svm_predictions <- predict(svm_model, newdata = test_dfm)
  cat(sprintf("Model with %.0f%% training and %.0f%% testing - Balanced Accuracy: %.3f\n",
              split*100, (1-split)*100, confusionMatrix(svm_predictions, as.factor(test_y))$byClass[11]))

}
```

```
## Model with 10% training and 90% testing - Balanced Accuracy: 0.630
## Model with 20% training and 80% testing - Balanced Accuracy: 0.685
## Model with 30% training and 70% testing - Balanced Accuracy: 0.720
## Model with 40% training and 60% testing - Balanced Accuracy: 0.738
## Model with 50% training and 50% testing - Balanced Accuracy: 0.727
## Model with 60% training and 40% testing - Balanced Accuracy: 0.714
## Model with 70% training and 30% testing - Balanced Accuracy: 0.736
## Model with 80% training and 20% testing - Balanced Accuracy: 0.764
## Model with 90% training and 10% testing - Balanced Accuracy: 0.833
```

The metric I used to evaluate the models is Balanced Accuracy (F-1 score is not an output in the confusion matrix function). It is a good metric for unbalanced data sets such as this one because it consists of the mean between sensitivity and specificity.

The model with the highest balanced accuracy was the 90/10 model (the one that used the most data to train).

c)

```
train_test_split <- 0.7*nrow(small_df)

train_dfm <- dfm(small_df$text[1:train_test_split], stem=TRUE, remove_punct=TRUE,
                 tolower=TRUE, remove = stopwords("english"))
train_y <- small_df$label[1:train_test_split]
test_dfm <- dfm(small_df$text[-(1:train_test_split)], stem=TRUE, remove_punct=TRUE,
                tolower=TRUE, remove = stopwords("english"))
test_y <- small_df$label[-(1:train_test_split)]

test_dfm <- dfm_match(test_dfm, features=featnames(train_dfm))

trainControl <- trainControl(method = "cv", number = 5)
logreg_model <- train(x=train_dfm, y=train_y, method="glmnet", family='binomial',
                      trControl=trainControl)

logreg_predictions <- predict(logreg_model, newdata = test_dfm)
confusionMatrix(logreg_predictions, as.factor(test_y), positive="positive")$byClass[11]
```

```
## Balanced Accuracy
##         0.7390873
```

When an algorithm does not converge, it means that it does not approach an specific value but it oscillates more and more. A potential solution would be to apply early stopping.

The balanced accuracy of the Logistic Regression model is 74.17%, compared to the 83% from the best SVM model.

QUESTION 7

a)

```
smaller_df <- raw_df[1:500, ]

train_test_split <- 0.8*nrow(smaller_df)


train_dfm <- dfm(smaller_df$text[1:train_test_split], stem=TRUE, remove_punct=TRUE,
                 tolower=TRUE, remove = stopwords("english"))
train_y <- smaller_df$label[1:train_test_split]
test_dfm <- dfm(smaller_df$text[-(1:train_test_split)], stem=TRUE, remove_punct=TRUE,
                tolower=TRUE, remove = stopwords("english"))
test_y <- smaller_df$label[-(1:train_test_split)]

test_dfm <- dfm_match(test_dfm, features=featnames(train_dfm))

train_df <- convert(train_dfm, to="data.frame")
test_df <- convert(test_dfm, to="data.frame")
```

b)

```
train_df <- convert(train_dfm, to="data.frame")
rf_model <- randomForest(x = train_df, y = as.factor(train_y), importance = TRUE)

sort(rf_model$importance[,'MeanDecreaseGini'], decreasing = TRUE)[1:10]
```

```
##    great    dirti   doc_id    locat     room    night     rude     love
## 3.398365 2.361724 1.839248 1.699033 1.582867 1.332148 1.217947 1.213308
##    excel     desk
## 1.190497 1.125760
```

c)

```
rf_predictions <- predict(rf_model, newdata = test_df)

accuracy_metrics(rf_predictions, test_y)
```

```
## Accuracy: 71.00%
## Precision: 69.74%
## Recall: 89.83%
## Specificity: 43.90%
## F1-Score: 0.79%
```

```
#                       PRED
#                  positive   negative
#               +-----------------------+
#      positive |  TP=53   |   FN=6    |
#TRUE           +-----------------------+
#      negative |  FP=27   |   TN=14   |
#               +-----------------------+
```

d)

```
rf_model1 <- randomForest(x = train_df, y = as.factor(train_y), importance = TRUE,
                          mtry = 0.5*sqrt(ncol(train_df)))
rf_model2 <- randomForest(x = train_df, y = as.factor(train_y), importance = TRUE,
                          mtry = 1.5*sqrt(ncol(train_df)))

rf_predictions1 <- predict(rf_model1, newdata = test_df)
rf_predictions2 <- predict(rf_model2, newdata = test_df)

cat(sprintf("First model(0.5*sqrt(features)):"))
```

```
## First model(0.5*sqrt(features)):
```

```
accuracy_metrics(rf_predictions1, test_y)
```

```
## Accuracy: 65.00%
## Precision: 65.38%
## Recall: 86.44%
## Specificity: 34.15%
## F1-Score: 0.74%
```

```
cat(sprintf("Second model(0.5*sqrt(features)):"))
```

```
## Second model(0.5*sqrt(features)):
```

```
accuracy_metrics(rf_predictions2, test_y)
```

```
## Accuracy: 73.00%
## Precision: 72.86%
## Recall: 86.44%
## Specificity: 53.66%
## F1-Score: 0.79%
```

It looks like the second model (the one using 1.5*sqrt(features)) is a little better, but the F-1 scores are very similar.