STOR767 - HW 4 Computing

Leo Li (PID: 730031954)

Overview / Instructions

This is the computing part of homework 4 of STOR 767. It will be **due on November 12, 2020 by 11:55 PM** on Sakai. You can directly edit this file to add your answers. Submit the Rmd file and the HTML version.

Problem: Yelp challenge 2019

Yelp has made their data available to public and launched Yelp challenge. More information. It is unlikely we will win the $5,000 prize posted but we still get to use their data for free. This exercise is designed for you to get hands on the whole process.

For this case study, we downloaded the data and took a 20k subset from **review.json**. *json* is another format for data. It is flexible and commonly-used for websites. Each item/subject/sample is contained in a brace *{}*. Data is stored as **key-value** pairs inside the brace. *Key* is the counterpart of column name in *csv* and *value* is the content/data. Both *key* and *value* are quoted. Each pair is separated by a comma. The following is an example of one item/subject/sample.

```
{
  "key1": "value1",
  "key2": "value2"
}
```

Data needed: yelp_review_20k.json available on Sakai

yelp_review_20k.json contains full review text data including the user_id that wrote the review and the business_id the review is written for. Here's an example of one review.

```
{
    // string, 22 character unique review id
    "review_id": "zdSx_SD6obEhz9VrW9uAWA",

    // string, 22 character unique user id, maps to the user in user.json
    "user_id": "Ha3iJu77CxlrFm-vQRs_8g",

    // string, 22 character business id, maps to business in business.json
    "business_id": "tnhfDv5Il8EaGSXZGiuQGg",

    // integer, star rating
    "stars": 4,

    // string, date formatted YYYY-MM-DD
    "date": "2016-03-09",

    // string, the review itself
```

```
    "text": "Great place to hang out after work: the prices are decent, and the ambience is fun. It's a

    // integer, number of useful votes received
    "useful": 0,

    // integer, number of funny votes received
    "funny": 0,

    // integer, number of cool votes received
    "cool": 0
}
```

Goals of the study

The goals are

1) Try to identify important words associated with positive ratings and negative ratings. Collectively we have a sentiment analysis.

2) To predict ratings using different methods.

1. JSON data and preprocessing data

i. Load *json* data

The *json* data provided is formatted as newline delimited JSON (ndjson). It is relatively new and useful for streaming.

```
{
  "key1": "value1",
  "key2": "value2"
}
{
  "key1": "value1",
  "key2": "value2"
}
```

The traditional JSON format is as follows.

```
[{
  "key1": "value1",
  "key2": "value2"
},
{
  "key1": "value1",
  "key2": "value2"
}]
```

We use `stream_in()` in the `jsonlite` package to load the JSON data (of ndjson format) as `data.frame`. (For the traditional JSON file, use `fromJSON()` function.)

```
pacman::p_load(jsonlite)
yelp_data <- jsonlite::stream_in(file("C:/Users/haoli/Desktop/Stor 767/computing/yelp_review_20k.json")
str(yelp_data)
```

```
## 'data.frame':    19999 obs. of  9 variables:
##  $ review_id  : chr  "Q1sbwvVQXV2734tPgoKj4Q" "GJXCdrto3ASJOqKeVWPi6Q" "2TzJjDVDEuAW6MR5Vuc1ug" "yi0
##  $ user_id    : chr  "hG7b0MtEbXx5QzbzE6C_VA" "yXQM5uF2jS6es16SJzNHfg" "n6-Gk65cPZL6Uz8qRm3NYw" "dac
```

```
##  $ business_id: chr  "ujmEBvifdJM6h6RLv4wQIg" "NZnhc2sEQy3RmzKTZnqtwQ" "WTqjgwHlXbSFevF32_DJVw" "ikC
##  $ stars      : num  1 5 5 5 1 4 3 1 2 3 ...
##  $ useful     : int  6 0 3 0 7 0 5 3 1 1 ...
##  $ funny      : int  1 0 0 0 0 0 4 1 0 0 ...
##  $ cool       : int  0 0 0 0 0 0 5 1 0 1 ...
##  $ text       : chr  "Total bill for this horrible service? Over $8Gs. These crooks actually had the
##  $ date       : chr  "2013-05-07 04:34:36" "2017-01-14 21:30:33" "2016-11-09 20:09:03" "2018-01-09 2
```

```
# different JSON format
# tmp_json <- toJSON(yelp_data[1:10,])
# fromJSON(tmp_json)
```

ii. Document term matrix (dtm)

Extract document term matrix for texts to keep words appearing at least .5% of the time among all 20000 documents. Go through the similar process of cleansing as we did in the lecture.

In this question, we would like to extract document term matrix for texts to keep words appearing at least .5% of the time among all 20,000 documents. In other words, we would like to keep words appearing in at least $20000 \times 0.005 = 100$ documents. The document term matrix is created as below.

```
corpus <- Corpus(VectorSource(yelp_data$text))
dtm <- DocumentTermMatrix(corpus,  control=list(
  tolower = T,
  removeNumbers = T,
  removePunctuation = T,
  stopwords = T,
  stripWhitespace = T,
  bounds = list(global = c(100,Inf))))
dtm
```

```
## <<DocumentTermMatrix (documents: 19999, terms: 1497)>>
## Non-/sparse entries: 677376/29261127
## Sparsity           : 98%
## Maximal term length: 15
## Weighting          : term frequency (tf)
```

a) Briefly explain what does this matrix record? What is the cell number at row 100 and column 405? What does it represent?

The matrix records the frequency of terms that occur in the documents. Specifically, each column represents one term (word), and each row represent one document, and each value contains the number of appearances of that term in that document.

The cell number at row 100 and column 405 is that,

```
as.matrix(dtm[100, 405])
```

```
##      Terms
## Docs  walking
##   100       0
```

It represents that the word "walking" appears 0 times in the 100th document.

b) What is the sparsity of the dtm obtained here? What does that mean?

According to the previous output, the sparsity of the dtm is 98%. It means that, within the document term matrix, 98% of the values of the document-word pairs equal to 0.

iii. Set the stars as a two category response variable called rating to be "1" = 5,4 and "0" = 1,2,3. Combine the variable rating with the dtm as a data frame called data2.

We dichotomize the response variable `stars` and we call the new response variable as `starsnew`. We also combine the variable rating with the dtm as a data frame called data2.

```
dtmtranspose <- as.matrix(dtm)
yelp_data$starsnew <- as.numeric(yelp_data$stars>3)
data2 <- cbind(yelp_data,dtmtranspose)
```

Analysis

Get a training data with 13000 reviews and the 5000 reserved as the testing data. Keep the rest (2000) as our validation data set.

We use the following program to split the data into a training set, a testing set, and a validation set.

```
N <- 20000
set.seed(1)
index.train <- sample(N, 13000)
training <- data2[index.train,]
rest <- data2[-index.train,]
set.seed(2)
index.test <- sample(N-13000, 5000)
testing <- rest[index.test,]
validate <- rest[-index.test,]
```

Then, we prepare the data ready for the analysis.

```
x_train <- as.data.frame(lapply(as.data.frame(training[,-c(1:10)]>0), as.numeric))
x_test <- as.data.frame(lapply(as.data.frame(testing[,-c(1:10)]>0), as.numeric))
x_validate <- as.data.frame(lapply(as.data.frame(validate[,-c(1:10)]>0), as.numeric))
y_train <- as.numeric(training$starsnew)
y_test <- as.numeric(testing$starsnew)
y_validate <- as.numeric(validate$starsnew)
```

2. LASSO

i. Use the training data to get Lasso fits. Choose lambda.1se. Keep the result here.

We use the training data to get Lasso fits.

```
set.seed(10)
fit.cv <- cv.glmnet(as.matrix(x_train), as.factor(y_train), alpha=1,
                    nfolds=10, family = "binomial")
coef.1se <- coef(fit.cv, s="lambda.1se")
coef.1se <- coef.1se[which(coef.1se !=0),]
```

ii. Feed the output from Lasso above, get a logistic regression.

We start by removing unneeded words from our training set, and then fit a logistic regression model.

```
train_lasso <- data.frame(y = y_train, x = x_train[,which(coef.1se[-1] !=0)])
result.glm.fit <- glm(y ~ ., data = train_lasso, family=binomial(link = "logit"))
```

a) Pull out all the positive coefficients and the corresponding words. Rank the coefficients in a decreasing order. Report the leading 2 words and the coefficients. Describe briefly the interpretation for those two coefficients.

Here, we pull out all the positive coefficients and the corresponding words and rank the coefficients in a descending order. The leading two words and their coefficients are printed below:

```
result.glm.coef <- as.numeric(coef(result.glm.fit))
result.glm.name <- names(coef.1se)
result.glm <- data.frame("name"=result.glm.name, "coef"=result.glm.coef)
result.glm <- result.glm[-1, ]
good.glm <- result.glm[result.glm$coef > 0, ]
good.word <- good.glm[order(-good.glm$coef),]
good.word[c(1,2),]
```

```
##     name     coef
## 18   get 1.669868
## 94 today 1.663514
```

The two coefficients are interpreted as follows:

1) The regression coefficient for the term "get" is estimated to be 1.67, which means that the log of the ratio of the odds of corresponding to a higher rating (4 or 5 stars) between the reviews with the word "get" and the reviews without the word "get" is estimated to be 1.67.

2) The regression coefficient for the term "today" is estimated to be 1.66, which means that the log of the ratio of the odds of corresponding to a higher rating (4 or 5 stars) between the reviews with the word "today" and the reviews without the word "today" is estimated to be 1.66.

b) Make a word cloud with the top 100 positive words according to their coefficients. Interpret the cloud briefly.

```
cor.special <- brewer.pal(8,"Dark2")
set.seed(10)
wordcloud(good.word$name[1:100], good.word$coef[1:100],
          scale=c(2.5,.01),min.freq=3,colors=cor.special, ordered.colors=F)
```

The above word cloud is interpreted as follows. The word cloud consists of the top 100 words that are associated with higher ratings (4 or 5 stars), the larger the association between the appearance of a particular word and the higher ratings (measured by the regression coefficient of the logisitc regression), the larger the font size of that word in the word cloud.

c) Repeat for the bag of negative words.

In this part, we pull out all the negative coefficients and the corresponding words and rank the coefficients in a descending order (in absolute value). The leading two words and their coefficients are printed below:

```
bad.glm <- result.glm[result.glm$coef < 0, ]
bad.word <- bad.glm[order(-bad.glm$coef, decreasing = T),]
bad.word[c(1,2),]
```

```
##       name      coef
## 8  another -2.797834
## 92  saying -2.323553
```

```
bad.word$neg_coef <- (-bad.word$coef)
```

The two coefficients are interpreted as follows:

1) The regression coefficient for the term "another" is estimated to be -2.80, which means that the log of the ratio of the odds of corresponding to a higher rating (4 or 5 stars) between the reviews with the word "another" and the reviews without the word "another" is estimated to be -2.80.

2) The regression coefficient for the term "saying" is estimated to be -2.32, which means that the log of the ratio of the odds of corresponding to a higher rating (4 or 5 stars) between the reviews with the word "saying" and the reviews without the word "saying" is estimated to be -2.32.

Now, we make a word cloud with the top 100 negative words according to their coefficients.

```
cor.special <- brewer.pal(8,"Dark2")
set.seed(10)
wordcloud(bad.word$name[1:100], bad.word$neg_coef[1:100],
          scale=c(3,.01),min.freq=3,colors=cor.special, ordered.colors=F)
```

The above word cloud is interpreted as follows. The word cloud consists of the top 100 words that are associated with lower ratings (1, 2, or 3 stars), the larger the association between the appearance of a particular word and the low ratings (measured by the regression coefficient of the logisitc regression), the larger the font size of that word in the word cloud.

d) Summarize the findings.

The findings of this part is summarized as follows. If we use Lasso to select important words that are strongly associated with higher or lower ratings, and also use lambda.1se as the tuning parameter. This results in selecting 488 variables. Then we conduct the logistic regression to model the logit of the probability that the reviews correspond to higher ratings using the 488 variables selected by Lasso. Then, from the logistic regression, there are 226 positive words according to their coefficients. Among those, the two leading words are "get" and "today", and their corresponding coefficients are estimated to be 1.67 and 1.66, respectively. There are also 262 negative words according to their coefficients, and among those, the two leading words are "another" and "saying", and their corresponding coefficients are estimated to be -2.80 and -2.32, respectively.

iii. Using majority votes find the testing errors

We would like to use majority votes to find the testing errors.

```
x_test_lasso <- data.frame(x = x_test[,which(coef.1se[-1] !=0)])
glm.predict <- predict(result.glm.fit, newdata=x_test_lasso, type="response")
glm.predict.label <- rep("0", nrow(x_test))
glm.predict.label[glm.predict > 0.5] ="1"
mean(as.character(y_test) != glm.predict.label)
```

```
## [1] 0.164
```

3. Random Forest

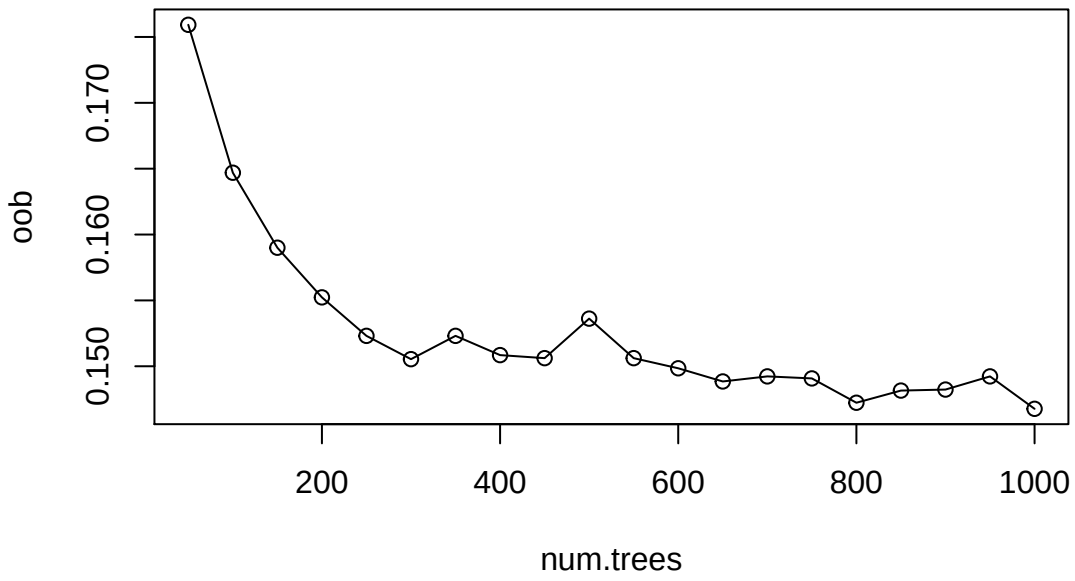
Now train the data using the training data set by RF. Get the testing error. Also explain how the RF works and how you tune the tuning parameters (`mtry` and `ntree`).

In this question, we train the data using the training data set by RF. In classification problem, the recommended `mtry` is $\sqrt{d} = \sqrt{1497} = 39$. We can start by setting `mtry = 39` and then tune the number of trees first. Here, we increase the `ntree` by 50 per iteration, and we record the OOB testing error for every iteration and we plot the graph of OOB testing error versus `ntree`.

```
train_rf <- data.frame(y = as.factor(y_train), x = x_train)
oob <- as.numeric()
num.trees <- as.numeric()
for(i in 1:20){
  set.seed(i*10)
  fit.rf.ranger <- ranger::ranger(y ~ ., data = train_rf, mtry = 39,
                                  num.trees = 50*i, importance="impurity")
oob <- c(oob, fit.rf.ranger$prediction.error)
num.trees <- c(num.trees, 50*i)
}
```

```
## Growing trees.. Progress: 94%. Estimated remaining time: 1 seconds.
## Growing trees.. Progress: 92%. Estimated remaining time: 2 seconds.
## Growing trees.. Progress: 76%. Estimated remaining time: 9 seconds.
## Growing trees.. Progress: 68%. Estimated remaining time: 14 seconds.
## Growing trees.. Progress: 67%. Estimated remaining time: 15 seconds.
## Growing trees.. Progress: 68%. Estimated remaining time: 14 seconds.
## Growing trees.. Progress: 68%. Estimated remaining time: 14 seconds.
## Growing trees.. Progress: 64%. Estimated remaining time: 17 seconds.
## Growing trees.. Progress: 59%. Estimated remaining time: 21 seconds.
## Growing trees.. Progress: 55%. Estimated remaining time: 25 seconds.
## Growing trees.. Progress: 52%. Estimated remaining time: 28 seconds.
## Growing trees.. Progress: 49%. Estimated remaining time: 32 seconds.
## Growing trees.. Progress: 99%. Estimated remaining time: 0 seconds.
## Growing trees.. Progress: 47%. Estimated remaining time: 35 seconds.
## Growing trees.. Progress: 94%. Estimated remaining time: 4 seconds.
## Growing trees.. Progress: 45%. Estimated remaining time: 38 seconds.
## Growing trees.. Progress: 89%. Estimated remaining time: 7 seconds.
```

```
plot(num.trees, oob)
lines(num.trees, oob)
```



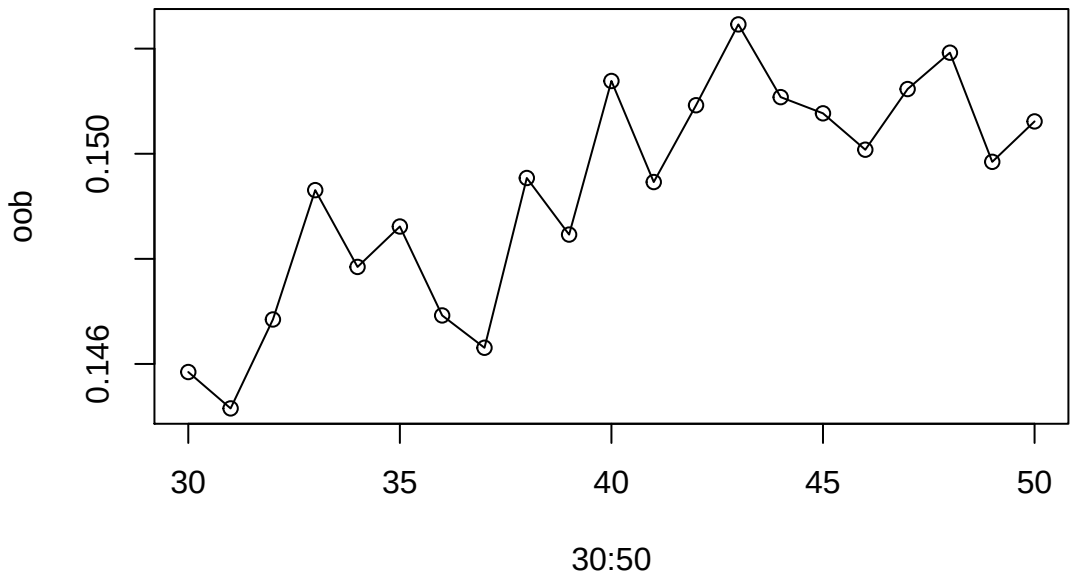
As we can see from the figure, any `ntree` beyond 700 would be sufficient to settle the OOB testing errors. Then, we fix `ntree=700`, and we would like to compare the OOB MSE to tune the `mtry`. Here we loop `mtry` from 30 to 50 and return the tesing OOB errors.

```
oob <- as.numeric()
for (p in 30:50)
{
set.seed(p*100)
fit.rf.ranger <- ranger::ranger(y ~ ., data = train_rf, mtry = p,
                                num.trees = 700, importance="impurity")
oob <- c(oob, fit.rf.ranger$prediction.error)
}
```

```
## Growing trees.. Progress: 79%. Estimated remaining time: 8 seconds.
## Growing trees.. Progress: 72%. Estimated remaining time: 12 seconds.
## Growing trees.. Progress: 67%. Estimated remaining time: 15 seconds.
## Growing trees.. Progress: 73%. Estimated remaining time: 11 seconds.
## Growing trees.. Progress: 71%. Estimated remaining time: 12 seconds.
## Growing trees.. Progress: 69%. Estimated remaining time: 14 seconds.
## Growing trees.. Progress: 56%. Estimated remaining time: 24 seconds.
## Growing trees.. Progress: 53%. Estimated remaining time: 27 seconds.
## Growing trees.. Progress: 47%. Estimated remaining time: 34 seconds.
## Growing trees.. Progress: 99%. Estimated remaining time: 0 seconds.
## Growing trees.. Progress: 65%. Estimated remaining time: 16 seconds.
## Growing trees.. Progress: 65%. Estimated remaining time: 16 seconds.
## Growing trees.. Progress: 64%. Estimated remaining time: 17 seconds.
## Growing trees.. Progress: 62%. Estimated remaining time: 18 seconds.
## Growing trees.. Progress: 62%. Estimated remaining time: 19 seconds.
## Growing trees.. Progress: 60%. Estimated remaining time: 20 seconds.
## Growing trees.. Progress: 59%. Estimated remaining time: 21 seconds.
```

```
## Growing trees.. Progress: 58%. Estimated remaining time: 22 seconds.
## Growing trees.. Progress: 57%. Estimated remaining time: 23 seconds.
## Growing trees.. Progress: 56%. Estimated remaining time: 24 seconds.
## Growing trees.. Progress: 55%. Estimated remaining time: 25 seconds.
## Growing trees.. Progress: 54%. Estimated remaining time: 26 seconds.
```

```
plot(30:50, oob)
lines(30:50, oob)
```



Then, we would observe that `mtry = 31` will lead to lowest OBB testing error, so that we end up choosing `ntree = 700`, and `mtry = 31`. Then we fit our final model:

```
fit.rf.ranger <- ranger::ranger(y ~ ., data = train_rf, mtry = 31,
                                num.trees = 700, importance="impurity")
```

```
## Growing trees.. Progress: 78%. Estimated remaining time: 8 seconds.
```

The way that the random forest works in classification can be described as follows. The random forest grows trees by including very many decision trees. It uses bagging and feature randomness when building each individual tree to try to create an uncorrelated forest of trees whose prediction by committee is more accurate than that of any individual tree. The binary prediction is then achieved by dichotomization based on the predicted probabilities.

The testing error is obtained as follows.

```
predict.rf <- predict(fit.rf.ranger, data=data.frame(x=x_test), type="response")
mean(as.character(y_test) != predict.rf$predictions)
```

```
## [1] 0.1396
```

4. Neural Network

Train a neural net with two layers with a reasonable number of neurons in each layer (say 20). You could try a different number of layers and different number of neutrons and see how the results change. Settle down on one final architecture. Report the testing errors.

In this question, we would like to train a neural net with two layers with 20 neurons in each layer.

```
defaultW <- getOption("warn")
options(warn = -1)
model <- keras_model_sequential() %>%
  layer_dense(units = 20, activation = "relu", input_shape = ncol(dtm)) %>%
  layer_dense(units = 20, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
options(warn = defaultW)
print(model)
```

```
## Model
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense (Dense)                (None, 20)                29960
## _________________________________________________________________________
## dense_1 (Dense)              (None, 20)                420
## _________________________________________________________________________
## dense_2 (Dense)              (None, 1)                 21
## =========================================================================
## Total params: 30,401
## Trainable params: 30,401
## Non-trainable params: 0
## _________________________________________________________________________
```

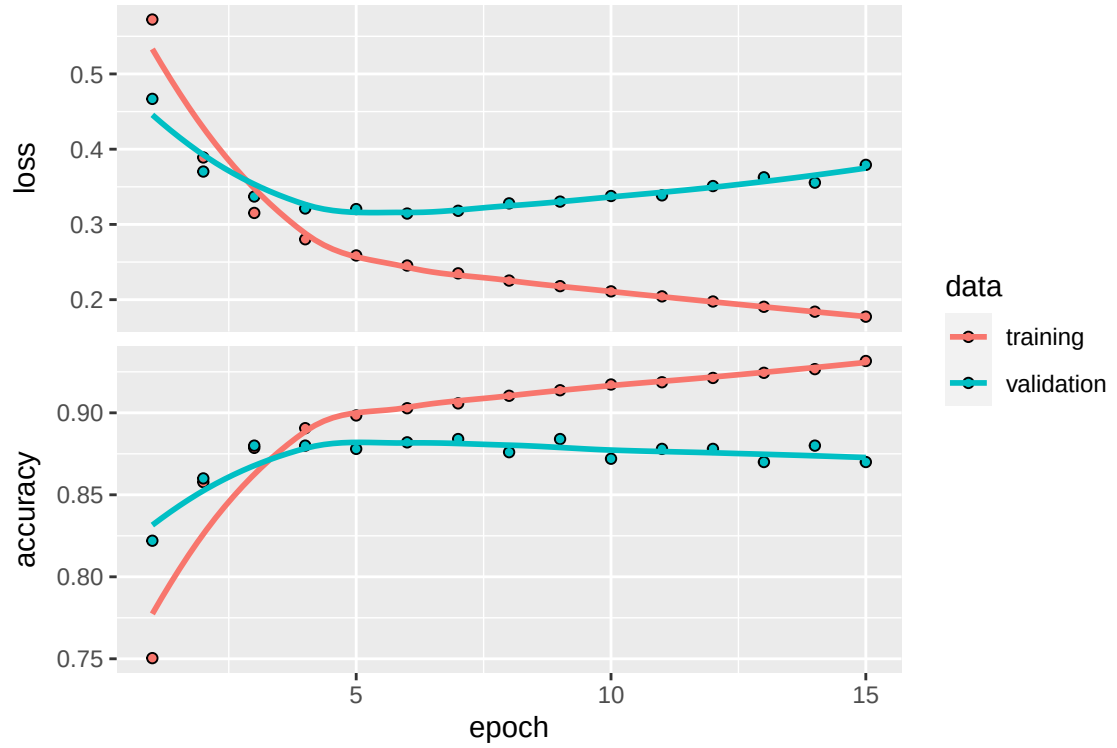
Then we compile our model as follows.

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

Next, we fit the neural network, and see what value of epochs will result in lower validation loss.

```
val_indices <- 1:500
x_val <- as.matrix(x_train[val_indices,])    # internal testing data
partial_x_train <- as.matrix(x_train[-val_indices,]) # training data
y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
set.seed(10)
fit1 <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 15,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
plot(fit1)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



From the graph above we see that by about 5 epochs our validation loss has bottomed out and we receive no further benefit from additional iterations. To avoid overfitting, we choose to use 5 epochs in our final model.

Now we are ready to fit our final model using all of the training data.

```
model %>% fit(as.matrix(x_train), y_train, epochs = 5, batch_size = 512)
```

Then, we use the testing data set to obtain the testing error.

```
results <- model %>% evaluate(as.matrix(x_test), y_test)
error <- 1-as.numeric(results[2])
print(paste0("the testing error is ", error))
```

```
## [1] "the testing error is 0.129800021648407"
```

As we can see, the testing error is already lower than that of Lasso & logistic regression and random forest. Now, we can check how the testing error will change if we add one more layer with 20 neurons.

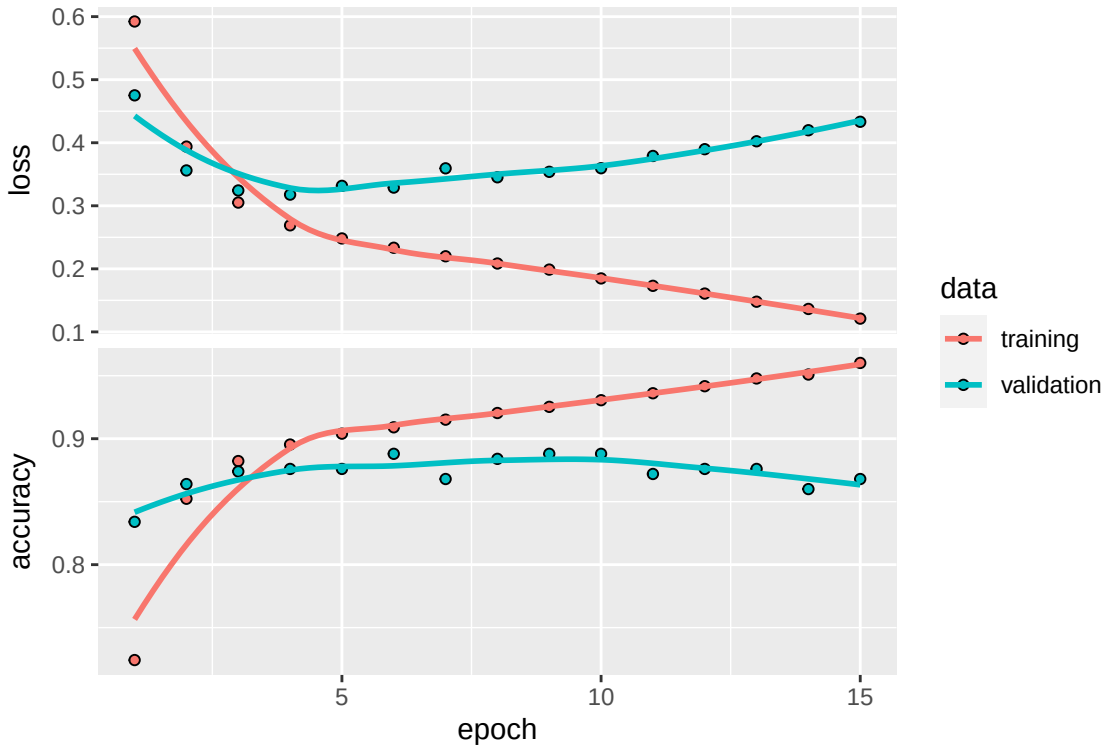
```
defaultW <- getOption("warn")
options(warn = -1)
model2 <- keras_model_sequential() %>%
  layer_dense(units = 20, activation = "relu", input_shape = ncol(dtm)) %>%
  layer_dense(units = 20, activation = "relu") %>%
  layer_dense(units = 20, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
options(warn = defaultW)
print(model2)
```

```
## Model
## Model: "sequential_1"
## _________________________________________________________________________
```

```
## Layer (type)                 Output Shape              Param #
## ================================================================================
## dense_3 (Dense)              (None, 20)                29960
## ________________________________________________________________________________
## dense_4 (Dense)              (None, 20)                420
## ________________________________________________________________________________
## dense_5 (Dense)              (None, 20)                420
## ________________________________________________________________________________
## dense_6 (Dense)              (None, 1)                 21
## ================================================================================
## Total params: 30,821
## Trainable params: 30,821
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
model2 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
set.seed(10)
fit2 <- model2 %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 15,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
plot(fit2)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



From the graph above we see that by about 4 epochs our validation loss has bottomed out and we receive no further benefit from additional iterations. To avoid overfitting, we choose to use 4 epochs in our final model.

Now we are ready to fit our final model using all of the training data.

```
model2 %>% fit(as.matrix(x_train), y_train, epochs = 4, batch_size = 512)
```

Then, we use the testing data set to obtain the testing error.

```
results <- model2 %>% evaluate(as.matrix(x_test), y_test)
error <- 1-as.numeric(results[2])
print(paste0("the testing error is ", error))
```

```
## [1] "the testing error is 0.134999990463257"
```

We can see that, even if we add one more layer, the testing error does not change very much, so we will just keep the original model with 2 layers and each with 20 neurons.

5. Final model

Which classifier(s) seem to produce the least testing error? Report the final model and accompany the validation error. Once again this is THE only time you use the validation data set. For the purpose of prediction, comment on how would you predict a rating if you are given a review using our final model?

According to the testing error that we computed in each part, we realize that neural network and random forest will result in pretty similar testing errors, and both of them are lower than that of Lasso & logistic regression. We can confirm this finding by using the validation data set.

The validation error of Lasso & logistic regression is,

```
x_val_lasso <- data.frame(x = x_validate[,which(coef.1se[-1] !=0)])
glm.predict <- predict(result.glm.fit, newdata=x_val_lasso, type="response")
```

```
glm.predict.label <- rep("0", nrow(x_validate))
glm.predict.label[glm.predict > 0.5] ="1"
mean(as.character(y_validate) != glm.predict.label)
```

```
## [1] 0.154077
```

The validation error of random forest is,

```
predict.rf <- predict(fit.rf.ranger, data=data.frame(x=x_validate), type="response")
mean(as.character(y_validate) != predict.rf$predictions)
```

```
## [1] 0.1405703
```

The validation error of neural network is,

```
results2 <- model %>% evaluate(as.matrix(x_validate), y_validate)
error2 <- 1-as.numeric(results2[2])
print(paste0("the validation error is ", error2))
```

```
## [1] "the validation error is 0.134567260742188"
```

Then, the validation errors that we obtained coincide with out previous findings. Therefore, we end up using the neural network to fit the model.

If I were given a review, I would predict the rating as follows:

1) Form a dtm matrix for the given review, based on the frequently appearing words (in the context of this question, the words that appears at least .5% of the time in the training data set) that we choose from the training data.
2) Prepare the data in the appropriate form for the classification method that we choose (Lasso & logistic regression, random forest, or neural network).
3) Based on the predicted probability, if the predicted probability is greater than 0.5, then we classify the review as high ratings (4 or 5 stars); and if the predicted probability is lower than 0.5, then we classify the review as low ratings (1, 2, or 3 stars).