IMDB Sentiment Analysis

Zachary Blanks

```
library(keras)
library(ggplot2)
library(purrr)
```

Overview

For our last project we are going to use neural networks, vector embeddings, and natural language processing (NLP) to perform sentiment analysis on an example data set provided by IMDB on $25,000$ movie reviews. Fortunately, this data, similar to MNIST, is a popular one and is available via Keras to introduce this concept. We are not going to achieve anywhere near state of the art performance, but we will use ideas that we've seen before and combine them with new NLP concepts to again show that neural networks can be used to tackle a wide range of unstructured data problems.

Unsurprisingly, we do have some data prepreprocessing to perform to ensure that our data is ready to run it in a sentiment analysis model. We will also have to do some basic data exploration to get an idea of how large we want our vocabulary to be for the embeddings and how long we want our reviews.

Data Exploration

Before we begin with our model, we should perform some simple exploratory analysis to get an idea about our data. One note before we begin – the data has already been pretty extensively pre-preprocessed. The reviews are not in their raw form, but rather have been converted into natural numbers which correspond to their frequency in the data. For example, if an integer is three, this means that this corresponded to the third most common word.

```
imdb <- dataset_imdb()
```

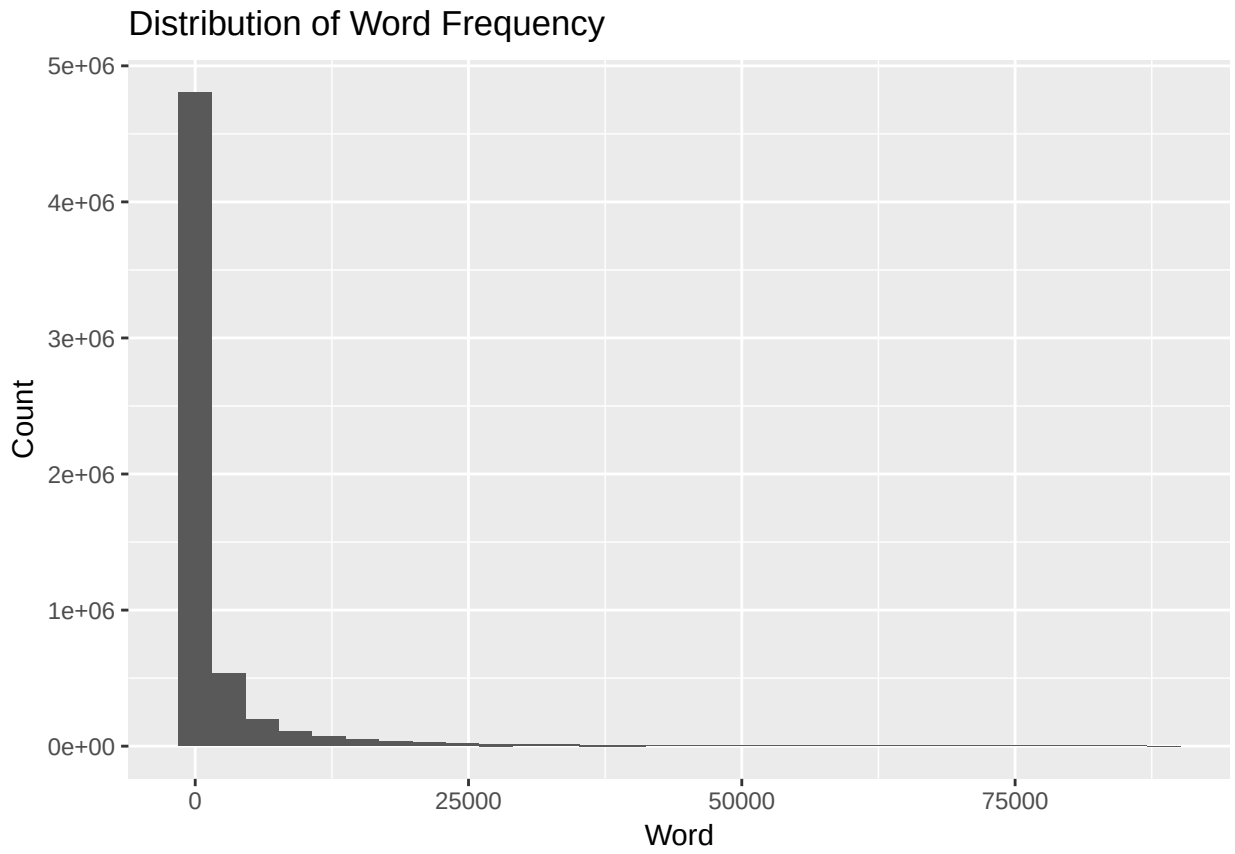
Now let's split our data into a training and test set

```
x_train <- imdb$train$x
y_train <- imdb$train$y
x_test <- imdb$test$x
y_test <- imdb$test$y
```

A good place to start is to see the distribution of words. A good way to simplify our model and speed up computation time is to decrease our vocabulary size.

```
df <- data.frame(unlist(x_train))
colnames(df) <- "vocab"

ggplot(data = df, mapping = aes(x = vocab)) +
  geom_histogram(bins = 30) + ggtitle(label = "Distribution of Word Frequency") +
  xlab(label = "Word") + ylab(label = "Count")
```



Unsurprisingly, a significant majority of the words used in the reviews can be described by roughly the first 5000 most popular words. Consequently before we train our model, we can reduce our vocabulary to include only those words. Let's also take a look at the distribution of review lengths.

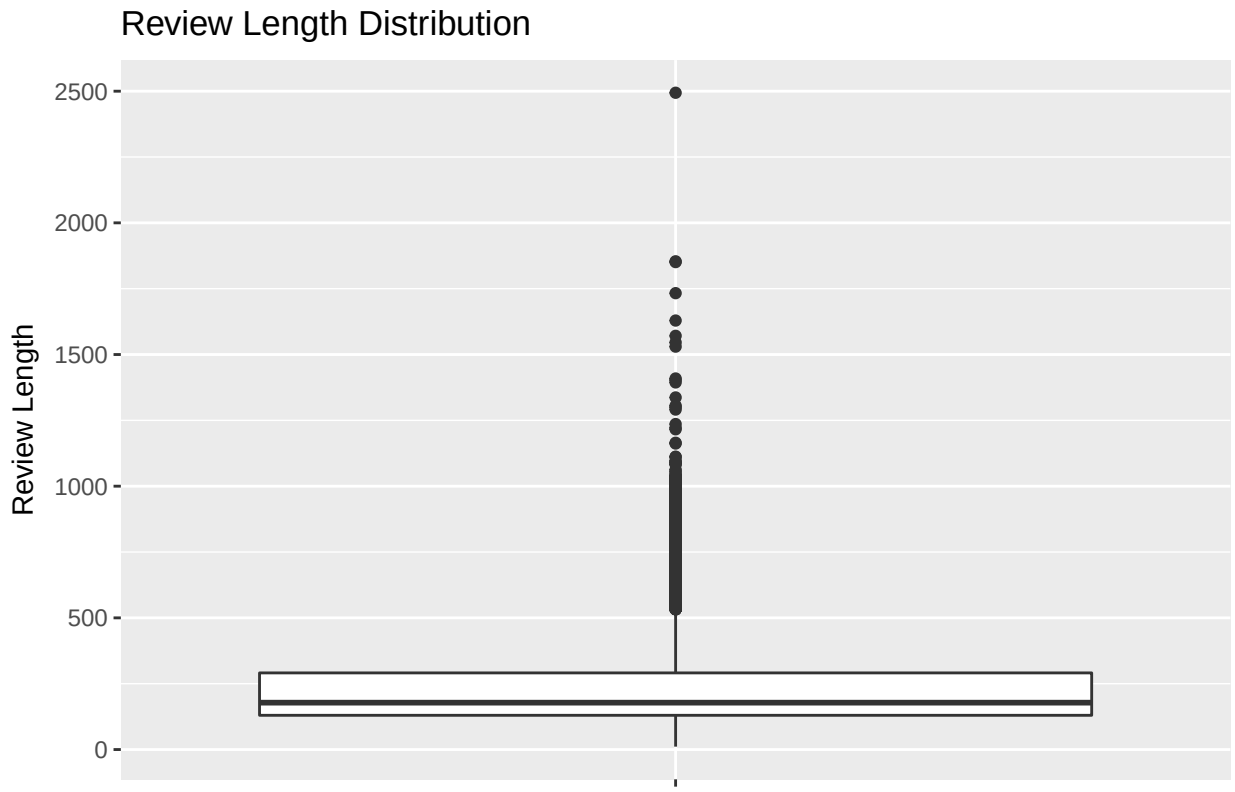
```
review_len <- map(.x = x_train, .f = length) %>%
  unlist()

summary(review_len)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    11.0   130.0   178.0   238.7   291.0  2494.0
```

Let's also visualize the distribution

```
df <- data.frame(review_len)
ggplot(data = df, mapping = aes(x = '', y = review_len)) +
  geom_boxplot() + ggtitle(label = 'Review Length Distribution') +
  xlab(label = '') + ylab(label = 'Review Length')
```



Looking at both our summary statistics and the box plot, it looks like a significant majority of the reviews are less than 500 words. Putting the word frequency and the review length pieces of information together, we're going to grab our data again, this time limiting ourselves to a vocabulary of 5000 words and the max length of a review is 500. If a particular review is less than 500 words, we will just pad it with an appropriate number of zeroes. Because this data is so common, Keras has already provided these functions for us, but if you were doing this in real life, these particular functions would not be too difficult to do yourself.

```
# Grab our data again with the constraints described above
imdb <- dataset_imdb(num_words = 5000, maxlen = 500, seed = 17)
x_train <- imdb$train$x
y_train <- imdb$train$y
x_test <- imdb$test$x
y_test <- imdb$test$y

# Pad our sequences if they're less than 500
x_train <- pad_sequences(sequences = x_train, maxlen = 500)
x_test <- pad_sequences(sequences = x_test, maxlen = 500)
```

Let's take a look at our data to make sure that everything looks good before we proceed.

```
dim(x_train)
```

```
## [1] 25000   500
```

```
x_train[1, ]
```

```
##   [1]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [15]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
```

```
##  [29]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [43]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [57]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [71]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [85]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [99]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [113]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [127]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [141]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [155]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [169]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [183]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [197]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [211]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [225]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [239]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [253]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [267]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [281]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [295]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [309]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [323]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [337]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [351]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [365]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [379]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [393]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [407]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [421]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## [435]    1    2  127   12  174   19   14  893  125    4 1986  794   56    8
## [449]    2   14 3198    9 1061   19    4    2    2 1538 2567    2  486    5
## [463] 1647 1733    2    2    6  932  767 3307 2249   11   14   20   15   82
## [477]  944   35 3149  136  121    2    2  180    6    2 2190   67   12   18
## [491]  919    5   18    6 1224    2  569    7    2 1236
```

Everything looks good! Since our data is in the correct form, we should be ready to implement an embedding and fully connected layer to perform sentiment analysis.

Sentiment Analysis

As you might imagine from our previous exercises, implementing complex models can be done in a matter of a few lines of codes using Keras. First we will define the Keras model.

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 5000, output_dim = 32,
                  input_length = 500) %>%
  layer_flatten() %>%
  layer_dense(units = 250, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1, activation = 'sigmoid')
```

Let's briefly discuss the arguments above. The only new things we've introduced are the word embedding layer and the flatten layer. For the embedding layer we have to provide it with both the vocabulary size so that it knows how many vector embeddings to compute, the length of the embedding vector (32 is a fairly

standard choice), and the length of the pieces of text. Second the reason we have to have the flatten layer is to ensure that we are providing a 2-d representation of our data. Other than those two new things, we're just doing the same thing as before. This is a relatively simple model, but we should get decent performance. Now we did to do our standard procedure of compiling and fitting our model.

```
model %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)

history <- model %>% fit(
  x = x_train, y = y_train,
  epochs = 5, batch_size = 128,
  validation_split = 0.2
)
```

```
model %>% evaluate(x_test, y_test)
```

```
## $loss
## [1] 0.4964783
##
## $acc
## [1] 0.8674751
```

That's not too bad. As you might imagine there a number of ways we can improve this and I encourage you to play around with the model to try to improve the accuracy.