Word Embeddings

Zachary Blanks

Before we begin, let's restart R one more time so that we have a fresh environment for this final module. We are also going to use ggplot and purrr for this section. Phil will go into much greater depth with the purrr module, but I will introduce it today to show some basic functionality.

```
library(keras)
library(tidyverse)
```

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
```

To start, let's see what a sample in our training set looks like.

```
X_train[[1]]
```

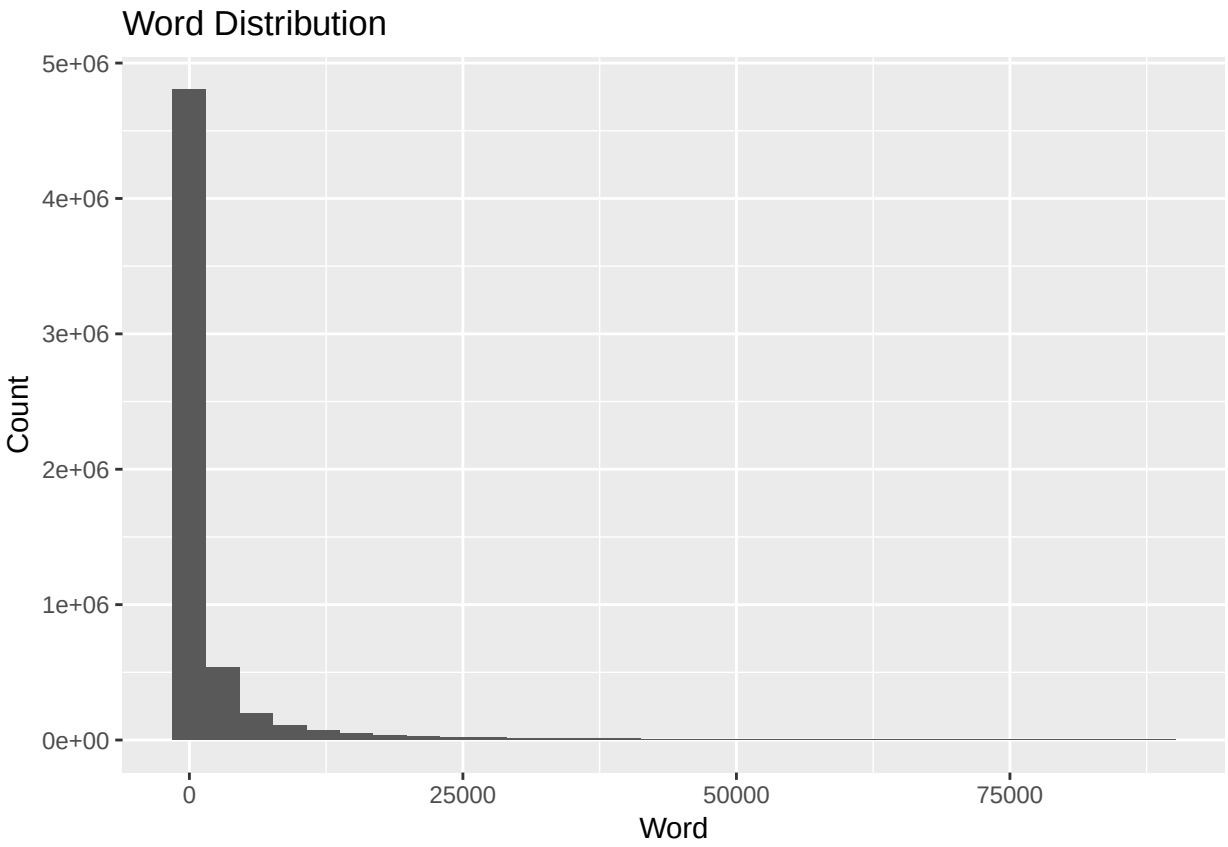
```
##   [1]     1    14    22    16    43   530   973  1622  1385    65   458
##  [12]  4468    66  3941     4   173    36   256     5    25   100    43
##  [23]   838   112    50   670 22665     9    35   480   284     5   150
##  [34]     4   172   112   167 21631   336   385    39     4   172  4536
##  [45]  1111    17   546    38    13   447     4   192    50    16     6
##  [56]   147  2025    19    14    22     4  1920  4613   469     4    22
##  [67]    71    87    12    16    43   530    38    76    15    13  1247
```

```
##  [78]     4    22    17   515    17    12    16   626    18 19193     5
##  [89]    62   386    12     8   316     8   106     5     4  2223  5244
## [100]    16   480    66  3785    33     4   130    12    16    38   619
## [111]     5    25   124    51    36   135    48    25  1415    33     6
## [122]    22    12   215    28    77    52     5    14   407    16    82
## [133] 10311     8     4   107   117  5952    15   256     4 31050     7
## [144]  3766     5   723    36    71    43   530   476    26   400   317
## [155]    46     7     4 12118  1029    13   104    88     4   381    15
## [166]   297    98    32  2071    56    26   141     6   194  7486    18
## [177]     4   226    22    21   134   476    26   480     5   144    30
## [188]  5535    18    51    36    28   224    92    25   104     4   226
## [199]    65    16    38  1334    88    12    16   283     5    16  4472
## [210]   113   103    32    15    16  5345    19   178    32
```

It seems that a sample is a numeric vector where each entry corresponds to the mapping for that particular word. For example, the second word in the review is the 14^{th} most common in the data.

Typically, natural language processing problems are very skewed – namely, a small number of words cover most of the uses in the data. If this is true, then this typically implies we can shrink the vocabulary without paying a big price in terms of model performance while significantly speeding up computation time (similar to PCA). To check this hypothesis, our first exercise for this project will be to look at the distribution of words in the data. Specifically, I would like you to create a histogram displaying the word usage distribution in each of the reviews. To do this, you will need to represent the training data as a DataFrame and use this DataFrame to make a histogram.

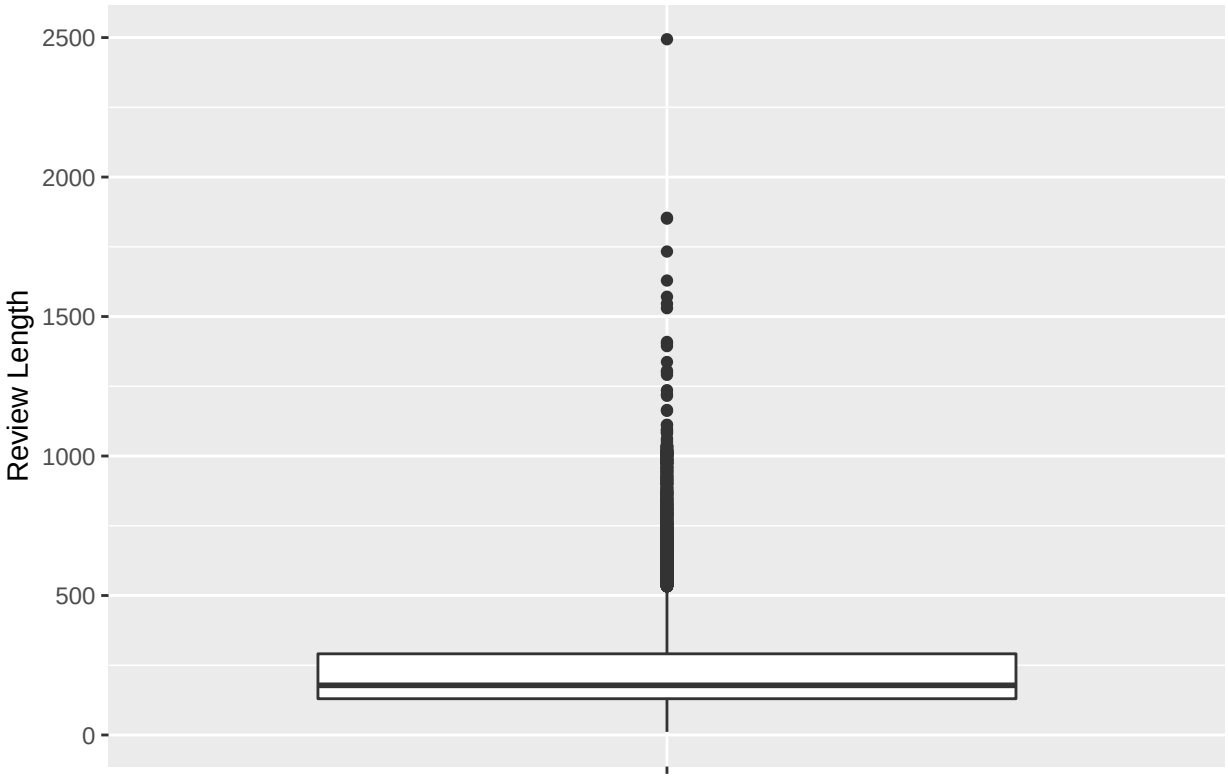
```r
create_word_df = function(x, sample_num) {
  # Get the number of times repeat the sample number
  sample_num_vect = rep(sample_num, length(x))

  # Return a DataFrame with two columns: the sample_num_vect and the words
  return(tibble(sample_num = sample_num_vect, word = x))
}

# Apply the user-created function to each element of the X_train list
# hence we will need purrr's map
word_df = map2_df(.x = X_train, .y = 1:length(X_train),
                  .f = ~ create_word_df(.x, .y))

# Using the DataFrame we just created to plot the distribution of words in
# the data
word_df %>%
  ggplot(aes(x = word)) +
  geom_histogram(bins = 30) +
  labs(x = "Word", y = "Count", title = "Word Distribution")
```



As we expected, 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.

Another element we need to determine before employing our sentiment analysis model is the distribution of review lengths. Our model will expect each review (which we will represent as a vector) to have the same length. Therefore we will need to determine a cut-off point for reviews so that if they are longer than this value we can fix this issue or if they're shorter than this value that we can pad the review with junk. Fortunately the way we created our word DataFrame is "tidy". This will allow us to easily answer this question. Therefore for the second exercise of this section, I want you to generate a boxplot displaying the distribution review lengths.

```
word_df %>%
  group_by(sample_num) %>%
  summarize(review_len = n()) %>%
  ggplot(aes(x = "", y = review_len)) +
  geom_boxplot() +
  labs(x = "", y = "Review Length")
```



Looking at 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. The only new layer we will need for this model is *layer_embedding* which will represent each word as a p-dimensional vector. We will also need a *layer_global_average_pooling_1d* to flatten our word emeddings into a two-dimensional space needed for the fully-connected layers. For our next exercise, I want you to make a word embedding network with the correct vocabulary dimesionality and where the words are represented as 32-dimesional vectors. Specifically I want you to create a network with an embedding layer, a global average pooling layer, a fully-connected layer with 64 nodes, and then a output layer which can predict binary targets. For binary outputs, you will need to use the "sigmoid" activation function.

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 5000, output_dim = 32,
                  input_length = 500) %>%
```

```
  layer_global_average_pooling_1d() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Now that we have defined the model, let's compile it using SGD and then train it.

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

res = model %>% fit(
  x = X_train, y = y_train,
  epochs = 5, batch_size = 128,
  validation_split = 0.25
)
```

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

```
## $loss
## [1] 0.6930963
##
## $acc
## [1] 0.5018857
```

Huh. It looks like the model did not do very well. It looks like the model is improving, but it's doing it very slowly. Why could that be?

Optimizers

Up to this point, we have just used vanilla SGD to as the solution method for our model, but in fact there are a huge number algorithms out there that minimize error for neural networks. For this lecture, we will focus on four: SGD, SGD with momentum, RMSProp, and Adam. If you are interested, there are full papers that provide more details on each of these algorithms.

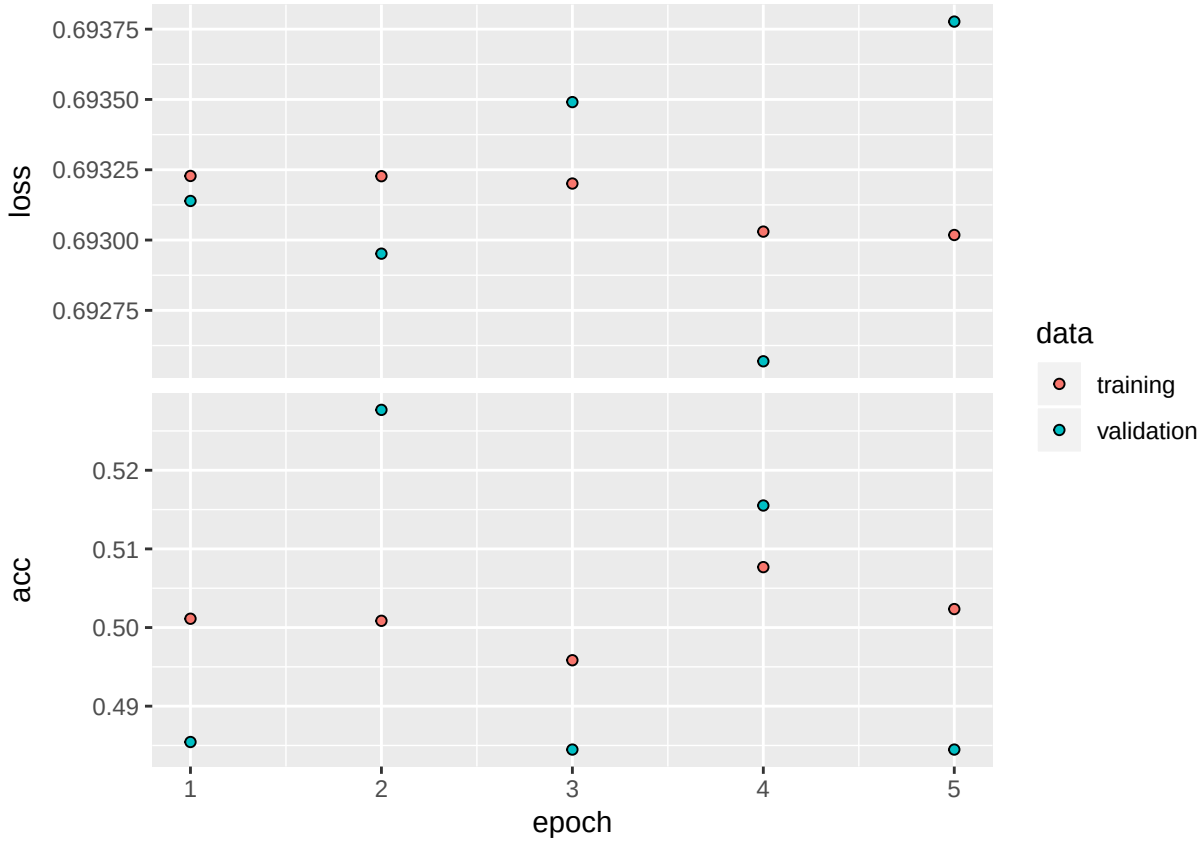
To see these algorithms in action, let's again break up into groups and have each third of the room apply SGD with momentum, RMSProp, and Adam. Use the same model we had above, but with the respective optimization algorithm. For SGD with momentum, set the momentum parameter equal to 0.9 and for the other algorithms, use the standard settings. Report the results.

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 5000, output_dim = 32,
                  input_length = 500) %>%
  layer_global_average_pooling_1d() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

# SGD with momentum
model1 = clone_model(model)

model1 %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_sgd(momentum = 0.9),
  metrics = c('accuracy')
```

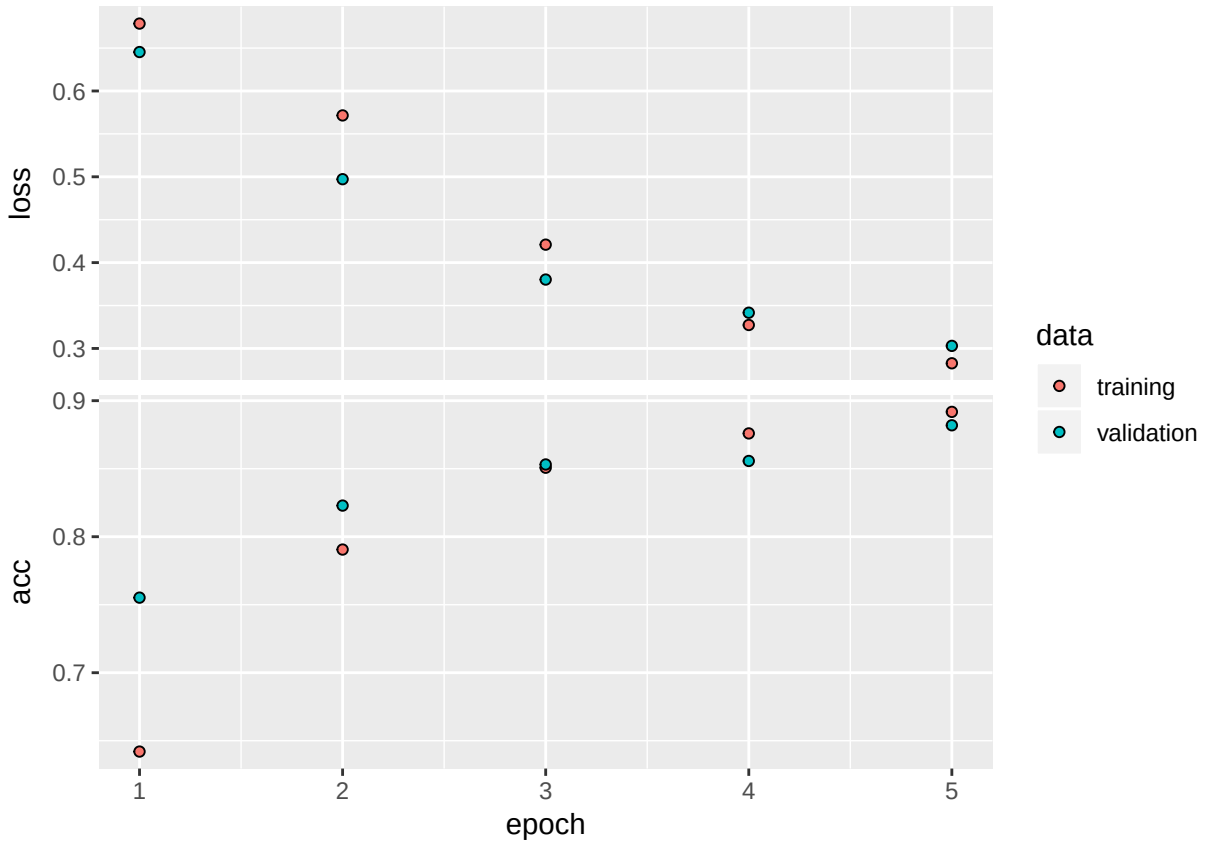
```
)

res1 = model1 %>% fit(
  x = X_train, y = y_train,
  epochs = 5, batch_size = 128,
  validation_split = 0.25,
  verbose = 0
)

# RMSProp
model2 = clone_model(model)

model2 %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)

res2 = model2 %>% fit(
  x = X_train, y = y_train,
  epochs = 5, batch_size = 128,
  validation_split = 0.25,
  verbose = 0
)

# Adam
model3 = clone_model(model)

model3 %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)

res3 = model3 %>% fit(
  x = X_train, y = y_train,
  epochs = 5, batch_size = 128,
  validation_split = 0.25,
  verbose = 0
)
```

Let's see how each of the model's do

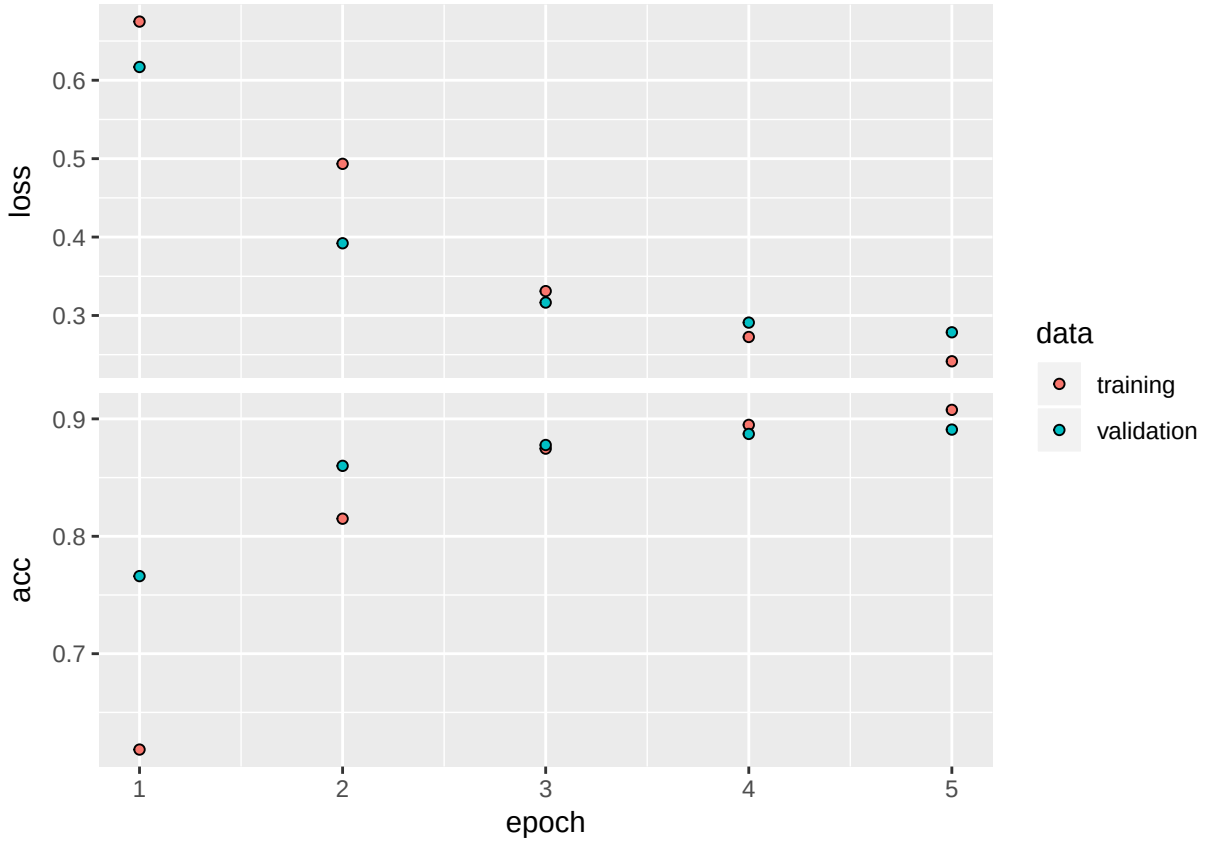
```
plot(res1)
```


loss
0.69375
0.69350
0.69325
0.69300
0.69275
acc
0.52
0.51
0.50
0.49
1
2
3
4
5
epoch
data
training
validation

```
plot(res2)
```


loss
0.6
0.5
0.4
0.3
acc
0.9
0.8
0.7
1
2
3
4
5
epoch
data
training
validation

```
plot(res3)
```




```
model1 %>% evaluate(X_test, y_test)
```

```
## $loss
## [1] 0.6931936
##
## $acc
## [1] 0.4984485
```

```
model2 %>% evaluate(X_test, y_test)
```

```
## $loss
## [1] 0.3077531
##
## $acc
## [1] 0.8761159
```

```
model3 %>% evaluate(X_test, y_test)
```

```
## $loss
## [1] 0.289547
##
## $acc
## [1] 0.8831336
```

It seems that the optimization algorithm can make a big difference in terms of performance. Typically RMSProp or Adam are good choices, but like almost everything with machine learning and especially with neural networks, try everything out and see what works best.

Vector Representation

In our previous model, we just arbitrarily chose the words to be represented by 32-dimensional vectors; let's see how sensitive our model is to that choice; using either a 4, 128, or 256 dimensional vector with the Adam optimizer, determine how sensitive the model is to this hyper-parameter

```
# Model with 4 dimensional vector
model1 <- keras_model_sequential() %>%
  layer_embedding(input_dim = 5000, output_dim = 4,
                  input_length = 500) %>%
  layer_global_average_pooling_1d() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model1 %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)

res1 = model1 %>% fit(
  x = X_train, y = y_train,
  epochs = 5, batch_size = 128,
  validation_split = 0.25,
  verbose = 0
)

# Model with 128-dimensional vector
model2 <- keras_model_sequential() %>%
  layer_embedding(input_dim = 5000, output_dim = 128,
                  input_length = 500) %>%
  layer_global_average_pooling_1d() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model2 %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)

res2 = model2 %>% fit(
  x = X_train, y = y_train,
  epochs = 5, batch_size = 128,
  validation_split = 0.25,
  verbose = 1
)

# Model with 256-dimesional vector
model3 = keras_model_sequential() %>%
  layer_embedding(input_dim = 5000, output_dim = 256,
                  input_length = 500) %>%
  layer_global_average_pooling_1d() %>%
  layer_dense(units = 64, activation = "relu") %>%
```

```
  layer_dense(units = 1, activation = "sigmoid")

model3 %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)

res3 = model3 %>% fit(
  x = X_train, y = y_train,
  epochs = 5, batch_size = 128,
  validation_split = 0.25,
  verbose = 1
)
```

Evaluate their performance out of sample

```
model1 %>% evaluate(X_test, y_test)
```

```
## $loss
## [1] 0.3410144
##
## $acc
## [1] 0.867857
```

```
model2 %>% evaluate(X_test, y_test)
```

```
## $loss
## [1] 0.2866004
##
## $acc
## [1] 0.8842316
```

```
model3 %>% evaluate(X_test, y_test)
```

```
## $loss
## [1] 0.2994927
##
## $acc
## [1] 0.8787416
```

I've now taught you everything you need to start using these techniques in practice. I would now like to switch gears to our Kaggle competition so that you have a chance to apply the techniques to a real dataset and compete against your classmates.