Test

Andreas Methling

26/11/2021

```
library(tidyverse)
library(lubridate)
library(magrittr)
library(FactoMineR)
library(factoextra)
library(uwot)
library(GGally)
library(rsample)
library(ggridges)
library(xgboost)
library(recipes)
library(parsnip)
library(glmnet)
library(tidymodels)
library(skimr)
library(VIM)
library(visdat)
library(ggmap)
library(ranger)
library(vip)
library(SnowballC)
library(tokenizers)
library(formatR)
```

```
## Warning: pakke 'formatR' blev bygget under R version 4.1.2
```

Data

```
library(readr)

data_start <- read_csv("C:/Users/andre/Desktop/lyrics-data.csv")
```

```
## Rows: 209522 Columns: 5
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (5): ALink, SName, SLink, Lyric, Idiom
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
artists_data <- read_csv("C:/Users/andre/Downloads/artists-data.csv")
```

```
## Rows: 3242 Columns: 6

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (4): Artist, Link, Genre, Genres
## dbl (2): Songs, Popularity

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Artist data

```
artists = artists_data %>%
  group_by(Artist) %>%
  count(Genre) %>%
  pivot_wider(names_from = Genre, values_from = n) %>%
  replace_na(list(Pop = 0, "Hip Hop" = 0, Rock = 0, "Funk Carioca" = 0, "Sertanejo" = 0, Samba = 0 )) %
  ungroup() %>%
  left_join(artists_data, by = c("Artist")) %>%
  select(-c(Genre, Genres, Popularity, Songs)) %>%
  distinct()
```

Data Rock or Pop

```
data_genre = data_start %>%
  filter(Idiom == "ENGLISH") %>%
  rename("Link" = "ALink") %>%
  inner_join(artists, by = c("Link")) %>%
  distinct() %>%
  mutate(name = paste(Artist, SName))%>%
  rename(text=Lyric) %>%
  filter(Pop==1 | Rock==1) %>%
  select(name, text, Pop, Rock) %>%
  distinct(name, .keep_all = T)



data_pop_rock=data_genre %>%
  mutate(genre = ifelse(Pop==1 & Rock == 1, "pop/rock", ifelse(Rock==1 & Pop==0, "Rock", ifelse(Rock ==
  select(-c(Pop, Rock))

data_pop_rock_labels= data_pop_rock %>%
  select(name, genre)
```

Data Rock and Pop

```
data = data_start %>%
  filter(Idiom == "ENGLISH") %>%
  rename("Link" = "ALink") %>%
  inner_join(artists, by = c("Link")) %>%
  distinct() %>%
  mutate(name = paste(Artist, SName))%>%
  rename(text=Lyric) %>%
  filter(Rock==1 & Pop==1) %>%
  select(name, text)%>%
  distinct(name, .keep_all = T)
```

Preprocessing / EDA

First we tokenize the data.

```
library(tidytext)
text_genre_tidy = data_pop_rock %>% unnest_tokens(word, text, token = "words")

head(text_genre_tidy)
```

```
## # A tibble: 6 x 3
##   name                         genre    word
##   <chr>                        <chr>    <chr>
## 1 10000 Maniacs More Than This pop/rock i
## 2 10000 Maniacs More Than This pop/rock could
## 3 10000 Maniacs More Than This pop/rock feel
## 4 10000 Maniacs More Than This pop/rock at
## 5 10000 Maniacs More Than This pop/rock the
## 6 10000 Maniacs More Than This pop/rock time
```

We remove short words and stopwords.

```
text_genre_tidy %<>%
  filter(str_length(word) > 2 ) %>%
  group_by(word) %>%
  ungroup() %>%
  anti_join(stop_words, by = 'word')
```

We use the hunspell package, which seems to produce the best stemming for our data. Reducing a word to its “root” word.

```
library(hunspell)
text_genre_tidy %>%
  mutate(stem = hunspell_stem(word)) %>%
  unnest(stem) %>%
  count(stem, sort = TRUE)
```

```
## # A tibble: 21,941 x 2
##    stem       n
##    <chr>  <int>
##  1 love  138187
##  2 time   70143
##  3 baby   62667
##  4 feel   62082
##  5 yeah   59708
##  6 gonna  44342
##  7 wanna  42810
##  8 girl   41029
##  9 day    39207
## 10 heart  39135
## # ... with 21,931 more rows
```

```
text_genre_tidy %<>%
  mutate(stem = hunspell_stem(word)) %>%
  unnest(stem) %>%
  select(-word) %>%
  rename(word = stem)
```

We weight the data using tf-idf (Term-frequency Inverse document frequency).

```
# TFIDF weights
text_tf_idf= text_genre_tidy %>%
group_by(name) %>%
  count(word, sort = TRUE) %>%
  ungroup() %>%
  bind_tf_idf(word, name, n) %>%
  arrange(desc(tf_idf))


text_genre_tf_idf = text_tf_idf %>%
  left_join(data_pop_rock_labels)
```

```
## Joining, by = "name"
```

We show the 25 most common words.

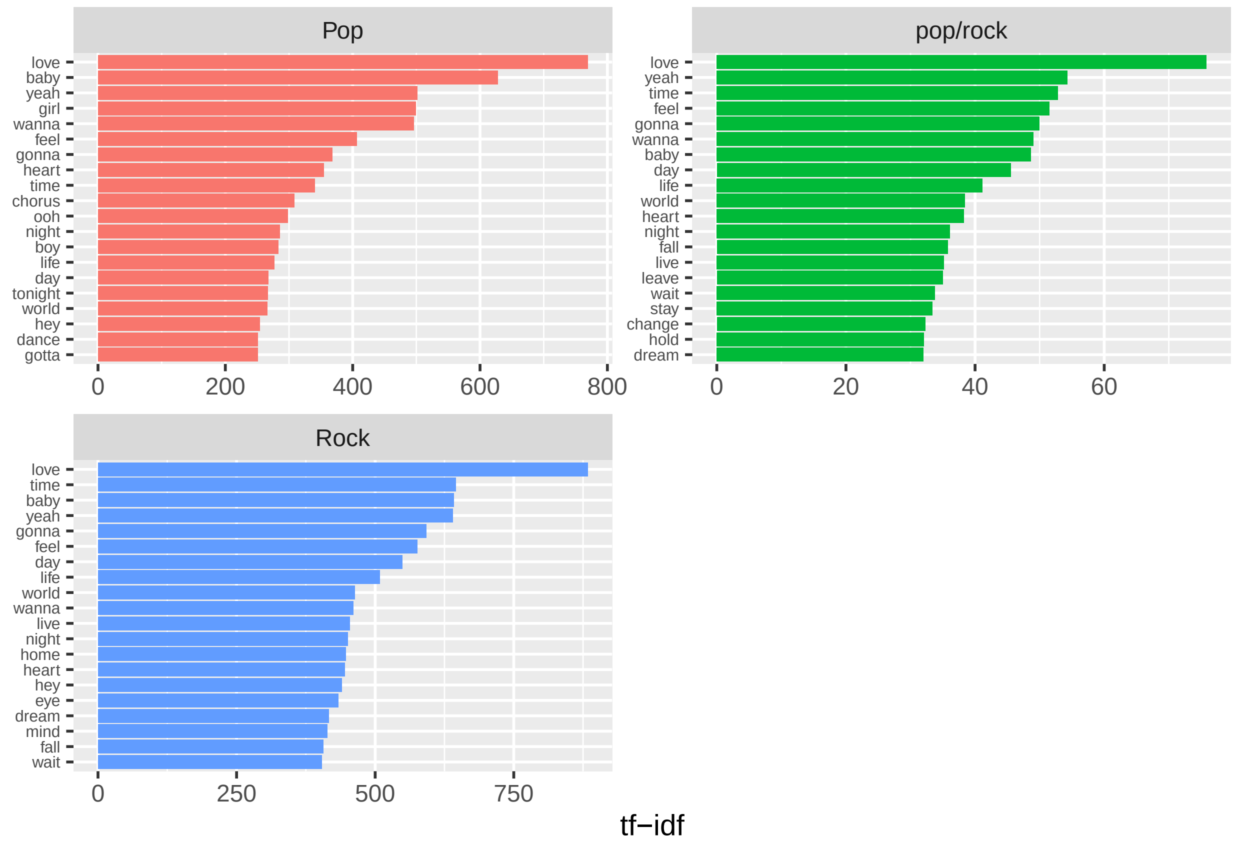
```
# TFIDF topwords
text_genre_tidy_rock= text_genre_tf_idf %>%
  filter(genre == "Rock")%>%
count(word, wt = tf_idf, sort = TRUE) %>% #remove
head(25)


text_genre_tidy_rock_pop= text_genre_tf_idf %>%
  filter(genre == "pop/rock")%>%
count(word, wt = tf_idf, sort = TRUE) %>% #remove
head(25)

text_genre_tidy_pop= text_genre_tf_idf %>%
  filter(genre == "Pop")%>%
count(word, wt = tf_idf, sort = TRUE) %>% #remove
head(25)
```

```
labels_words <- text_genre_tf_idf %>%
group_by(genre) %>%
count(word, wt = tf_idf, sort = TRUE, name = "tf_idf") %>%
dplyr::slice(1:20) %>% #slice
ungroup()
```

```
labels_words %>%
mutate(word = reorder_within(word, by = tf_idf, within = genre)) %>% #Pop & Rock
ggplot(aes(x = word, y = tf_idf, fill = genre)) +
geom_col(show.legend = FALSE) +
labs(x = NULL, y = "tf-idf") +
facet_wrap(~genre, ncol = 2, scales = "free") +
coord_flip() +
scale_x_reordered() +
theme(axis.text.y = element_text(size = 6))
```



```
text_tidy = data %>% unnest_tokens(word, text, token = "words")
head(text_tidy)
```

```
## # A tibble: 6 x 2
##   name                          word
##   <chr>                         <chr>
## 1 10000 Maniacs More Than This i
## 2 10000 Maniacs More Than This could
## 3 10000 Maniacs More Than This feel
## 4 10000 Maniacs More Than This at
## 5 10000 Maniacs More Than This the
## 6 10000 Maniacs More Than This time
```

```
text_tidy %<>%
  filter(str_length(word) > 2 ) %>%
  group_by(word) %>%
  ungroup() %>%
  anti_join(stop_words, by = 'word')
```

We use stemming

```
text_tidy %<>%
  mutate(stem = hunspell_stem(word)) %>%
  unnest(stem) %>%
   select(-word) %>%
  rename(word = stem)
```

```
top_10000_words=text_tidy %>%
  count(word,sort = T) %>%
  head(10000) %>%
  select(word)
data_top_10000=top_10000_words %>%
  left_join(text_tidy, by= c("word"))
```

Bing

```
sentiment_bing= data_top_10000 %>%
  inner_join(get_sentiments("bing")) %>%
  mutate(sentiment= ifelse(sentiment == "positive", 1,0))
```

```
## Joining, by = "word"
```

```
sentiment_bing %<>%
  group_by(name) %>%
  summarise(mean= mean(sentiment))%>%
  mutate(label= ifelse(mean>=0.5, 1,0))
```

Afinn

```
sentiment_afinn= data_top_10000 %>%
  inner_join(get_sentiments("afinn"))
```

```
## Joining, by = "word"
```

```
sentiment_afinn %<>%
  group_by(name) %>%
  summarise(mean= mean(value))%>%
  mutate(label= ifelse(mean>=0, 1,0))
```

Data

```
data_bing= sentiment_bing %>%
  inner_join(data)%>%
  select(text, label, name)
```

```
## Joining, by = "name"
```

```
data_afinn= sentiment_afinn %>%
  inner_join(data)%>%
  select(text, label, name)
```

```
## Joining, by = "name"
```

Neural Network Bing

Using our bing data set

```
data_bing_n= data_bing %>%
  rename( y = label )
```

We start by creating test and training data

```
library(rsample)
split= initial_split(data_bing_n, prop = 0.75)
train_data= training(split)
test_data= testing(split)
```

```
x_train_data= train_data %>% pull(text)
y_train_data= train_data %>% pull(y)
x_test_data= test_data %>% pull(text)
y_test_data= test_data %>% pull(y)
```

Now it is time to load keras and make some adjustments to the data. The data are lyrics so not a lot of special characters are used but we still remove them just to be sure. And then we tokenize our data as we know from basic machine learning to get like a bag of words from our song lyrics and lastly we create a list where every song has a vector which includes the words as a numerical character if the words contained in the tweets are among the 100000 most used words in the data set.

```
library(keras)
```

```
##
## Vedhæfter pakke: 'keras'

## De følgende objekter er maskerede fra 'package:vip':
##
##     metric_accuracy, metric_auc

## Det følgende objekt er maskeret fra 'package:yardstick':
##
##     get_weights
```

```r
#for training data
tokenizer_train <- text_tokenizer(num_words = 5000,
                                  filters = "!\"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n" ) %>%
  fit_text_tokenizer(x_train_data)
sequences_train = texts_to_sequences(tokenizer_train, x_train_data)
#For test data
tokenizer_test <- text_tokenizer(num_words = 5000,
                                 filters = "!\"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n" ) %>%
  fit_text_tokenizer(x_test_data)
sequences_test = texts_to_sequences(tokenizer_test, x_test_data)
```

Baseline model

One-hot encoding

we use this function Daniel made to vectorize the sequences :)

```r
vectorize_sequences <- function(sequences, dimension) {
  results <- matrix(0, nrow = length(sequences), ncol = dimension)
  for(i in 1:length(sequences)){
    results[i, sequences[[i]]] <- 1
  }
  return(results)
}
```

we use it on the training and test data

```r
x_train <- sequences_train %>% vectorize_sequences(dimension = 5000)
x_test <- sequences_test %>% vectorize_sequences(dimension = 5000)
str(x_train[1,])
```

```
##  num [1:5000] 1 1 1 1 1 0 0 1 1 1 ...
```

What the above has done to the data is, that every tweet now is a row and every feature/word now is a column and then if the tweets has e.g. word 1 then it would have the value 1 otherwise zero. So we basically now have a matrix of size [2488x5000] [number of song in training set x number of words].

The model

The above data is then used in our baseline model with an input shape of 5000 because that is the size of our input. Then we run it through two dense "relu" layers which is normal procedure for a baseline model. Lastly we have a dense layer with the output which is of unit 1 and is a "sigmoid" layer which means it returns a value between 0 and 1 as we want, as we wanna figure out if a song is positive or negative.

```
model_keras <- keras_model_sequential()
model <- model_keras %>%
  layer_dense(units = 128, activation = "relu", input_shape = c(5000)) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

We use baseline model compiling with optimizer "adam", loss "binary" as we are dealing with a binary case and the metric we wanna maximize is accuracy.

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

Here the structure of the model can be viewed, where it can be seen that the model has 656769 tunable parameters, so not the biggest of models but not the smallest either.

```
summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                       Output Shape                      Param #
## ================================================================================
## dense_2 (Dense)                    (None, 128)                       640128
## ________________________________________________________________________________
## dense_1 (Dense)                    (None, 128)                       16512
## ________________________________________________________________________________
## dense (Dense)                      (None, 1)                         129
## ================================================================================
## Total params: 656,769
## Trainable params: 656,769
## Non-trainable params: 0
## ________________________________________________________________________________
```

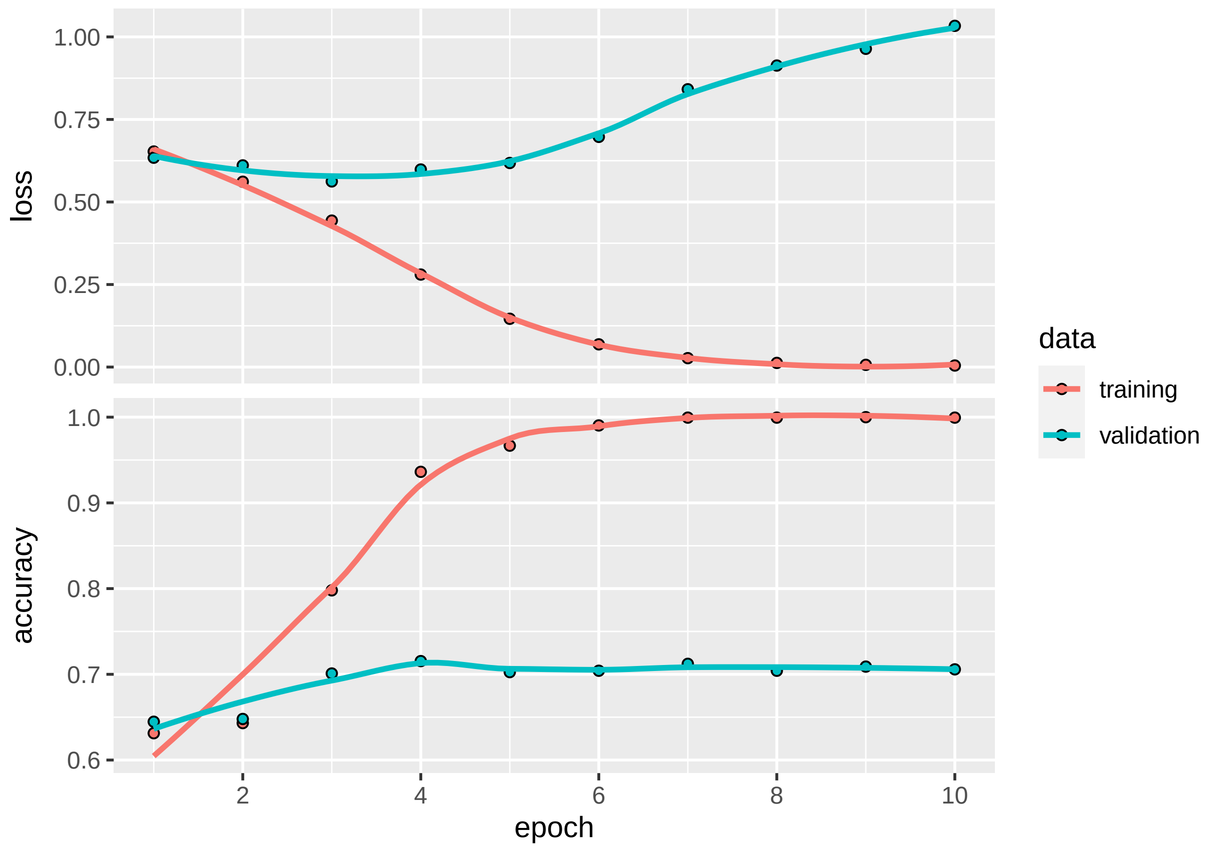
And now the model is run 10 times with a batch size of 256

```
set.seed(476)
history_ann <- model %>% fit(
  x_train,
  y_train_data,
  epochs = 10,
  batch_size = 256,
  validation_split = 0.25
)
```

We then plot the result of the model

```
plot(history_ann)
```

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



The top graph shows the lose of the model, where the blue line is the loss of the validation set, which initially falls a bit but then it rises back up. The lower graph shows the same, that the moment the loss rises in the validation set the accuracy falls again. The training set does a lot better than the validation set, which is a indicator of, that our model is over fitted.

Running our model on our test data also shows a bad result

```
metrics = model %>% evaluate(x_test, y_test_data); metrics
```

```
##      loss  accuracy
## 1.9356257 0.5481928
```

We will now try to tune the model to get a better result and prevent the over fitting.

Model tunning

We introduce some dropout layers and reduce the weights of each layer to minimize the number of parameters in the model to prevent the overfitting we saw above.

```
model_keras <- keras_model_sequential()
model2 <- model_keras %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(5000)) %>%
  layer_dropout(rate = 0.1) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dropout(rate = 0.1) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

We use baseline model compiling with optimizer “adam”, loss “binary” as we are dealing with a binary case and the metric we wanna maximize is accuracy.

```
model2 %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = "accuracy"
)
```

Here the structure of the model can be viewed, where it can be seen that the model has 656769 tunable parameters, so not the biggest of models but not the smallest either.

```
summary(model2)
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## dense_5 (Dense)                     (None, 16)                      80016
## ________________________________________________________________________________
## dropout_1 (Dropout)                 (None, 16)                      0
## ________________________________________________________________________________
## dense_4 (Dense)                     (None, 16)                      272
## ________________________________________________________________________________
## dropout (Dropout)                   (None, 16)                      0
## ________________________________________________________________________________
## dense_3 (Dense)                     (None, 1)                       17
## ================================================================================
## Total params: 80,305
## Trainable params: 80,305
## Non-trainable params: 0
## ________________________________________________________________________________
```

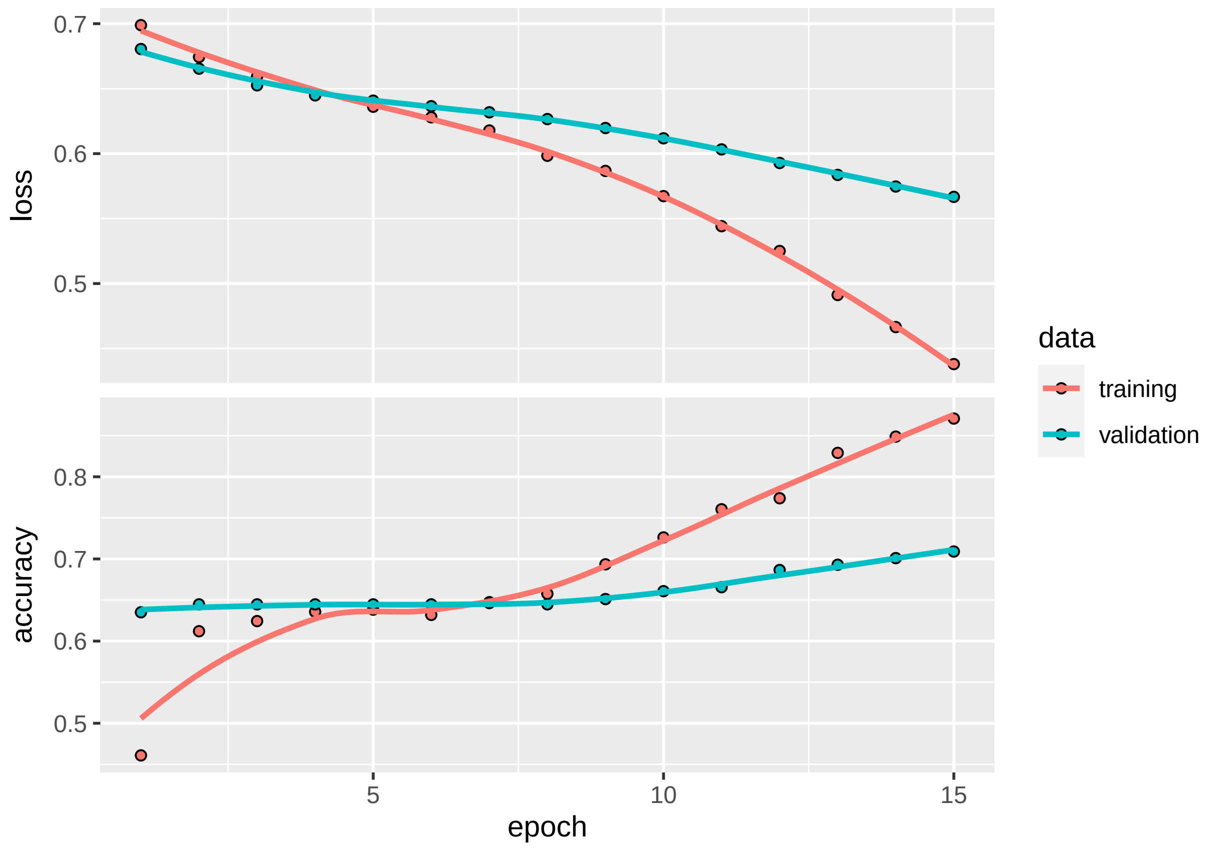
And now the model is run 10 times with a batch size of 512, so a bigger batch size than the baseline model.

```
set.seed(476)
history_ann2 <- model2 %>% fit(
  x_train,
  y_train_data,
  epochs = 15,
  batch_size = 512,
  validation_split = 0.25
)
```

We then plot the result of the tunned model

```
plot(history_ann2)
```

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



The model we have tried to tune seems better as we can see the loss function both of the validation data and the training data continues to go down after every epoch. The accuracy also increases in every period until it stalls out a bit but still this model looks better than the baseline model.

```
metrics2 = model2 %>% evaluate(x_test, y_test_data); metrics2
```

```
##      loss  accuracy
## 0.6887822 0.5638554
```

The accuracy of this model is better than the baseline model, but with an accuracy of 59% it is still not any good.

Rnn model with paded data

Padding

In our first baseline model, we used a document-term matrix as inputs for training, with one-hot-encodings (= dummy variables) for the 10.000 most popular terms. This has a couple of disadvantages. Besides being a very large and sparse vector for every review, as a “bag-of-words”, it did not take the word-order (sequence) into account.

This time, we use a different approach, therefore also need a different input data-structure. We now use pad_sequences() to create a integer tensor of shape (samples, word_indices). However, song vary in length, which is a problem since Keras requieres the inputs to have the same shape across the whole sample. Therefore, we use the maxlen = 300 argument, to restrict ourselves to the first 300 words in every song.

The data is paded

```
x_train_pad <- sequences_train %>% pad_sequences(maxlen=300)
x_test_pad <- sequences_test %>% pad_sequences(maxlen=300)
```

```
glimpse(x_train_pad)
```

```
##  num [1:2488, 1:300] 0 0 0 0 933 0 0 0 84 0 ...
```

Now if the value in e.g. the first column of the first tweet is 0 it means that the first word in the first tweet is not one of the 100000 most used words and there for our model has no integer for it. If there is an integer e.g. “386” it means that that the 386 most commonly used word is the first word in the tweet.

The model setting up the model we will first use a layer_embedding to compress our initial one-hot-encoding vector of length 5000 to a “meaning-vector” (=embedding) of the lower dimensionality of 32. Then we add a layer_simple_rnn on top, and finally a layer_dense for the binary prediction of review sentiment.

```
model_keras2 <- keras_model_sequential()
model_rnn <- model_keras2 %>%
  layer_embedding(input_dim = 5000, output_dim = 32) %>%
  layer_simple_rnn(units = 32, activation = "tanh") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Here the structure of the model can be seen

```
summary(model_rnn)
```

```
## Model: "sequential_2"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## embedding (Embedding)               (None, None, 32)                160000
## ________________________________________________________________________________
## simple_rnn (SimpleRNN)              (None, 32)                      2080
## ________________________________________________________________________________
## dense_6 (Dense)                     (None, 1)                       33
## ================================================================================
## Total params: 162,113
## Trainable params: 162,113
## Non-trainable params: 0
## ________________________________________________________________________________
```

Again we use a basic setup for binary prediction.

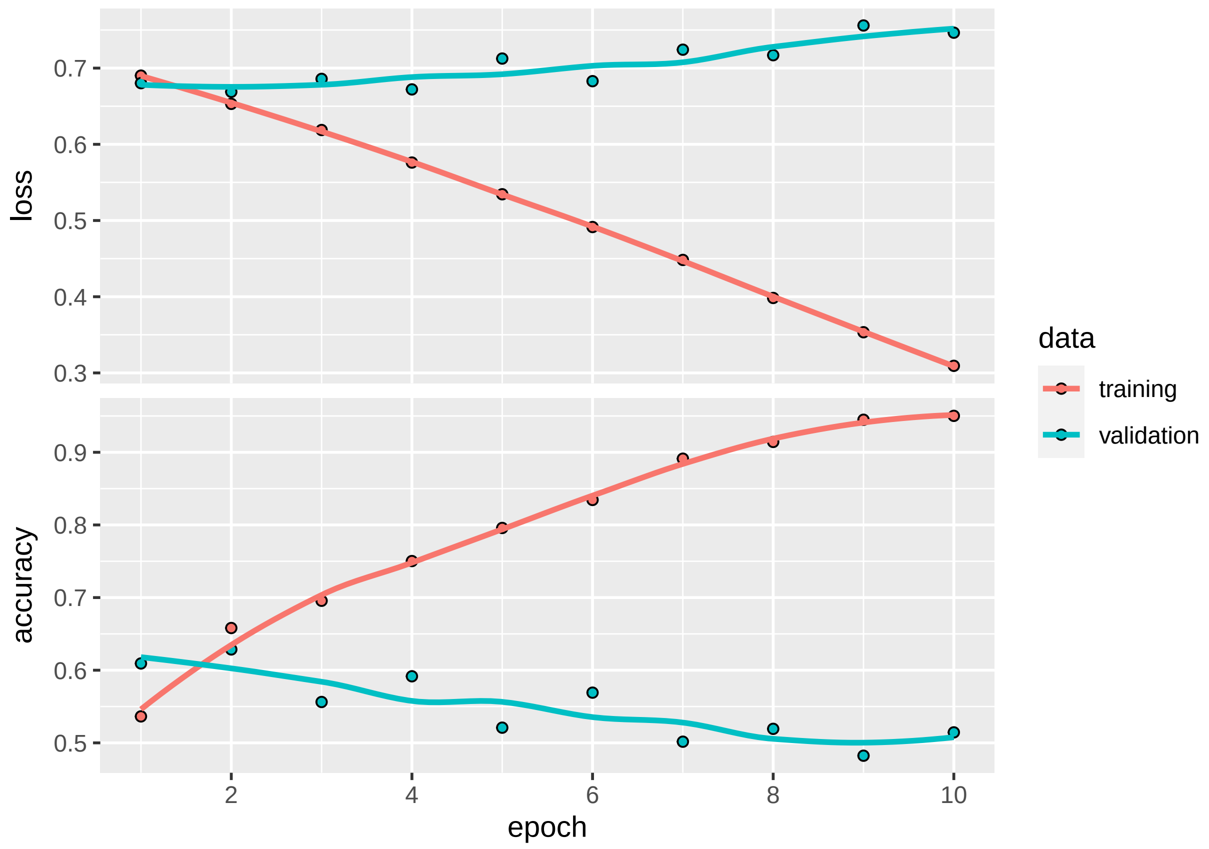
```r
model_rnn %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = "accuracy"
)
```

And run our model

```r
set.seed(476)
history_rnn <- model_rnn %>% fit(
  x_train_pad, y_train_data,
  epochs = 10,
  batch_size = 516,
  validation_split = 0.25
)
```

```r
plot(history_rnn)
```

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



Again the traning set outperforms the validation set a lot, which shows our model is overfitted. Further we see that the loss of the validation set starts to climb after a couple of epochs and the accuracy to fall, so not a good model.

```
metrics3 = model_rnn %>% evaluate(x_test_pad, y_test_data); metrics3
```

```
##      loss  accuracy 
## 0.7293488 0.5421687
```

Running our model on the test data shows an accuracy of 52%, but we will now try to fine tune it to make it better.

Tunning the model This time we again try to reduce the number of parameters to prevent over fitting. We also make another rnn layer and drop a fraction of the units with a drop_out input. Return_sequences = TRUE return the full state sequence of the first rnn layer so next rnn layer gets the full sequence of the input.xMethl

```
model_keras2 <- keras_model_sequential()
model_rnn2 <- model_keras2 %>%
  layer_embedding(input_dim = 5000, output_dim = 16) %>%
  layer_simple_rnn(units = 16, return_sequences = TRUE, activation = "tanh",recurrent_dropout=0.1) %>%
  layer_simple_rnn(units = 16, return_sequences = FALSE, activation = "tanh", recurrent_dropout=0.1) %>
  layer_dense(units = 1, activation = "sigmoid")
```

Here the structure of the model can be seen

```
summary(model_rnn2)
```

```
## Model: "sequential_3"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #     
## ================================================================================
## embedding_1 (Embedding)             (None, None, 16)                80000       
## ________________________________________________________________________________
## simple_rnn_2 (SimpleRNN)            (None, None, 16)                528         
## ________________________________________________________________________________
## simple_rnn_1 (SimpleRNN)            (None, 16)                      528         
## ________________________________________________________________________________
## dense_7 (Dense)                     (None, 1)                       17          
## ================================================================================
## Total params: 81,073
## Trainable params: 81,073
## Non-trainable params: 0
## ________________________________________________________________________________
```

Again we use a basic setup for binary prediction.

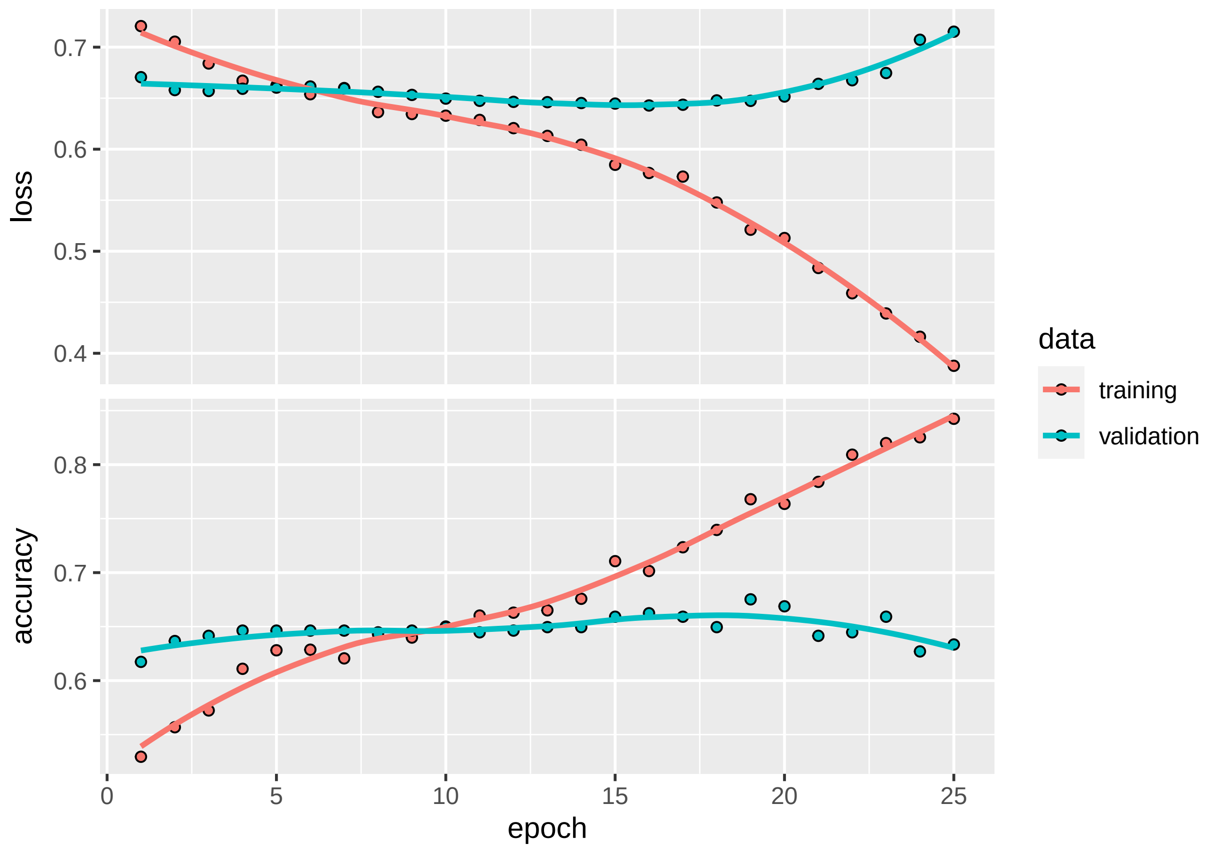
```
model_rnn2 %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = "accuracy"
)
```

And run our model

```
set.seed(476)
history_rnn2 <- model_rnn2 %>% fit(
  x_train_pad, y_train_data,
  epochs = 25,
  batch_size = 516,
  validation_split = 0.25
)
```

```
plot(history_rnn2)
```

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



The loss of the traning and validation data seems to follow each other quiet well for the first couple of epochs. After that the validation set begin to vary a lot flying up and down. The accuracy also follows each other a lot, but after around 12 epocs they cross and the traning data runs of.

```
metrics4 = model_rnn2 %>% evaluate(x_test_pad, y_test_data); metrics4
```

```
##      loss  accuracy
## 0.8689529 0.5361446
```

Running our model on the test data shows an accuracy of 58%, this is better than the baseline model but now at all good.

LTSM

We will now try our data on a LSTM model, but here we are only running one model, since it takes a very long time to run it. We start by using an embeding layer and then we go to a LSTM layer with a unit size of our paded data, which have the size of 300 due to it being the 300 first words in every song. In the lstm layer, we freeze some of the input weigths and also some of the state weights and since we only use one layer we set sequence = false, so it just compiles the input to a single output.

```
model_lstm <- keras_model_sequential() %>%
  layer_embedding(input_dim = 5000, output_dim = 32) %>%
  layer_lstm(units = 300, dropout = 0.25, recurrent_dropout = 0.25, return_sequences = FALSE) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

We use a base compiling

```
model_lstm %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = c("acc")
)
```

The model has a lot of parameters which makes it time consumeing to run it.

```
summary(model_lstm)
```

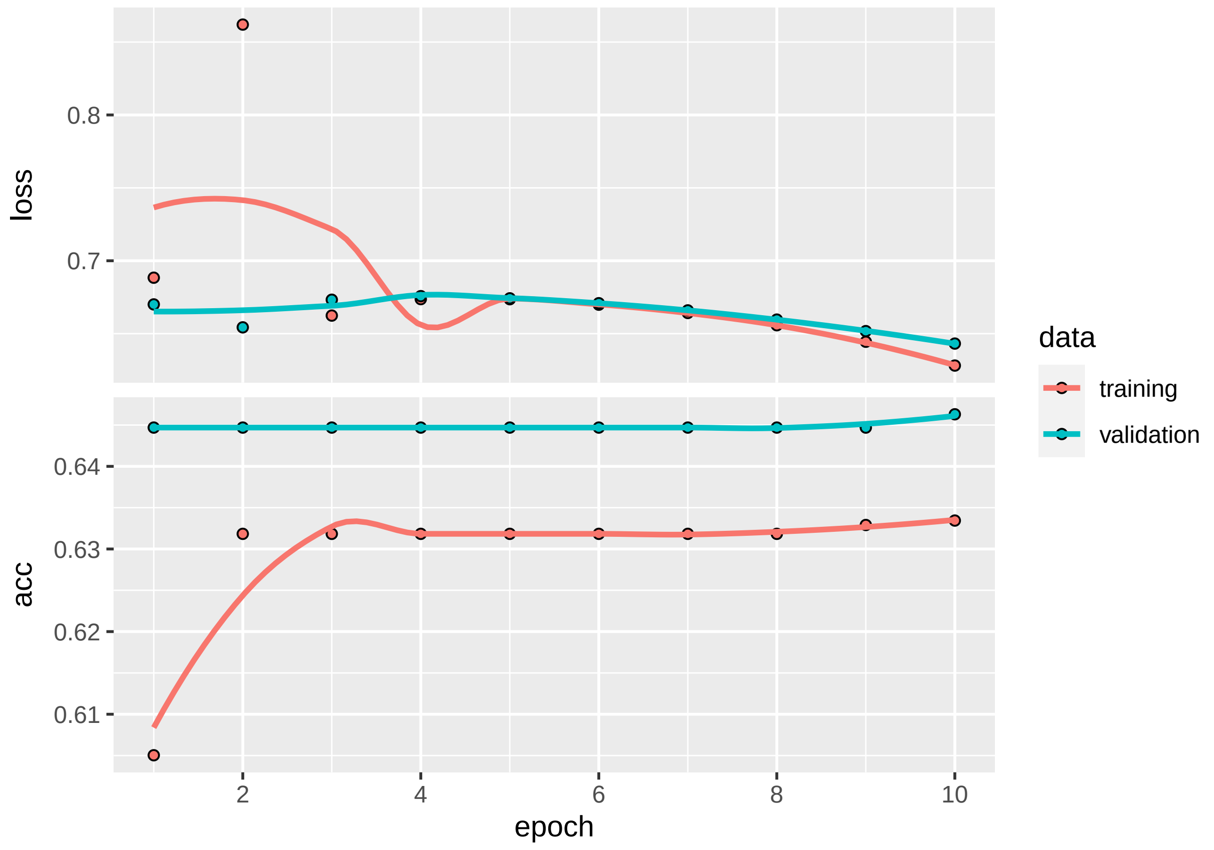
```
## Model: "sequential_4"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                        Param #
## ================================================================================
## embedding_2 (Embedding)             (None, None, 32)                    160000
## ________________________________________________________________________________
## lstm (LSTM)                         (None, 300)                         399600
## ________________________________________________________________________________
## dense_8 (Dense)                     (None, 1)                           301
## ================================================================================
## Total params: 559,901
## Trainable params: 559,901
## Non-trainable params: 0
## ________________________________________________________________________________
```

Then we run the mode

```
set.seed(476)
history_lstm <- model_lstm %>% fit(
  x_train_pad, y_train_data,
  epochs = 10,
  batch_size = 512,
  validation_split = 0.25
)
```

```
plot(history_lstm)
```

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



The modeldoesnt seem to perform any better than the previous ones, but the training and validation data did follow each other quiet well for some epochs, but then at the end they split up due to the increasing loss value of the validation data.

```
metrics5 = model_lstm %>% evaluate(x_test_pad, y_test_data); metrics5
```

```
##      loss       acc
## 0.6862711 0.5879518
```

A rly poor result to say the least with an accuracy of 63%.

NN multiclass

```
library(magrittr)

sentiment_nrc <- text_tidy %>%
  inner_join(get_sentiments("nrc"))
```

```
## Joining, by = "word"
```

```
multi_data=sentiment_nrc %>%
  filter(sentiment %in% c("negative", "positive", "joy", "fear")) %>%
  count(name, sentiment) %>%
  pivot_wider(names_from = sentiment, values_from = n, values_fill = 0) %>%
  mutate(label= pmax(positive, joy, negative, fear)) %>%
  mutate(label= ifelse(label == fear, "fear", ifelse(label == positive, "positive", ifelse(label == nega
  select(name, label) %>%
  inner_join(data)
```

```
## Joining, by = "name"
```

```
multi_data_new=sentiment_nrc %>%
  filter(sentiment %in% c("trust", "sadness", "joy", "fear")) %>%
  count(name, sentiment) %>%
  pivot_wider(names_from = sentiment, values_from = n, values_fill = 0) %>%
  mutate(label= pmax(trust, joy, sadness, fear)) %>%
  mutate(label= ifelse(label == fear, "fear", ifelse(label == trust, "trust", ifelse(label == sadness,
  select(name, label) %>%
  rename(y= label)%>%
  inner_join(data)
```

```
## Joining, by = "name"
```

```
library(rsample)
split5= initial_split(multi_data_new, prop = 0.75)
train_data5= training(split5)
test_data5= testing(split5)
```

```
x_train_data5= train_data5 %>% pull(text)
x_test_data5= test_data5 %>% pull(text)
```

```
y_train_data5= train_data5 %>% select('y') %>% mutate(y= recode(y, "joy" = 0, "sadness" = 1,"fear" =2,
y_test_data5= test_data5 %>% select('y') %>% mutate(y= recode(y, "joy" = 0, "sadness" = 1,"fear" =2, "t
```

```
to_one_hot <- function(labels, dimension = 4) {
  results <- matrix(0, nrow = length(labels), ncol = dimension)
  for (i in 1:length(labels))
    results[i, labels[[i]]] <- 1
  results
}
one_hot_train_labels <- to_one_hot(y_train_data5)
one_hot_test_labels <- to_one_hot(y_test_data5)
```

```
library(keras)
# For Training data
tokenizer10 <- text_tokenizer(num_words = 10000) %>%
  fit_text_tokenizer(x_train_data5)
sequences10 <- texts_to_sequences(tokenizer10, x_train_data5)
tokenizer11 <- text_tokenizer(num_words = 10000) %>%
  fit_text_tokenizer(x_test_data5)
sequences11 <- texts_to_sequences(tokenizer11, x_test_data5)
```

```r
vectorize_sequences <- function(sequences, dimension) {
  results <- matrix(0, nrow = length(sequences), ncol = dimension)
  for (i in 1:length(sequences))
    results[i, sequences[[i]]] <- 1
  results
}
x_train10 <- sequences10 %>% vectorize_sequences(dimension = 10000)
x_test10 <- sequences11 %>%  vectorize_sequences(dimension = 10000)
```

```r
model_keras5 <- keras_model_sequential()
model5 <- model_keras5 %>%
  layer_dense(units = 16, activation = "relu", input_shape = ncol(x_train10)) %>%
    layer_dense(units = 16, activation = "relu") %>%
    layer_dense(units = ncol(one_hot_train_labels), activation = "softmax")
```

We use baseline model compiling with optimizer “adam”, loss “binary” as we are dealing with a binary case and the metric we wanna maximize is accuracy.

```r
model5 %>% compile(
  optimizer = "adam",
  loss = "categorical_crossentropy",
  metrics = "accuracy"
)
```

Here the structure of the model can be viewed, where it can be seen that the model has 656769 tunable parameters, so not the biggest of models but not the smallest either.

```r
summary(model5)
```

```
## Model: "sequential_5"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## dense_11 (Dense)                    (None, 16)                      160016
## ________________________________________________________________________________
## dense_10 (Dense)                    (None, 16)                      272
## ________________________________________________________________________________
## dense_9 (Dense)                     (None, 4)                       68
## ================================================================================
## Total params: 160,356
## Trainable params: 160,356
## Non-trainable params: 0
## ________________________________________________________________________________
```

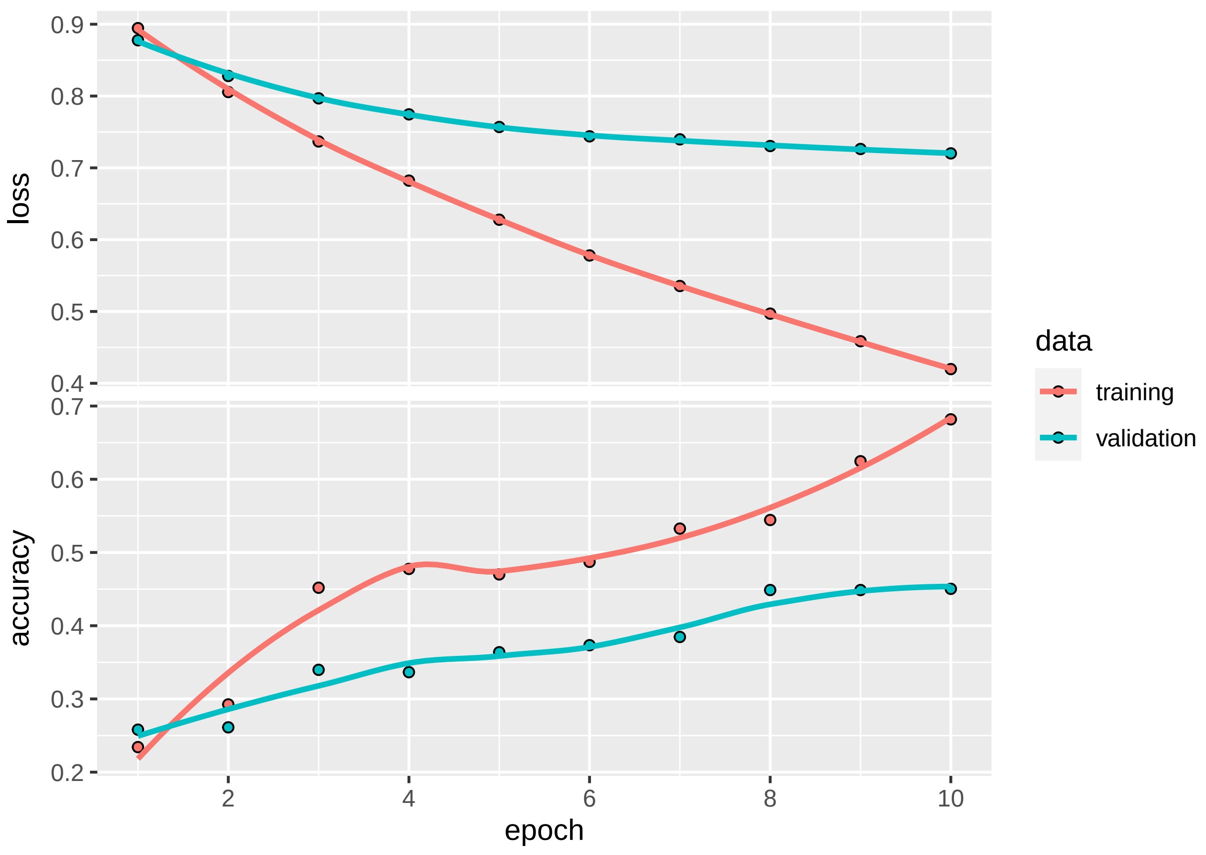
And now the model is run 10 times with a batch size of 256

```r
set.seed(476)
history_ann5 <- model5 %>% fit(
  x_train10,
  one_hot_train_labels,
  epochs = 10,
  batch_size = 256,
  validation_split = 0.25
)
```

We then plot the result of the model

```
plot(history_ann5)
```

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



```
metrics10 = model5 %>% evaluate(x_test10, one_hot_test_labels); metrics10
```

```
##      loss  accuracy
## 0.7892295 0.3870192
```