Sentiment del

Simon

30/11/2021

```
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 == 0 & Pop == 1, "Pop", 0)))) %>%
  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) %>%
  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 within the 3 genres.

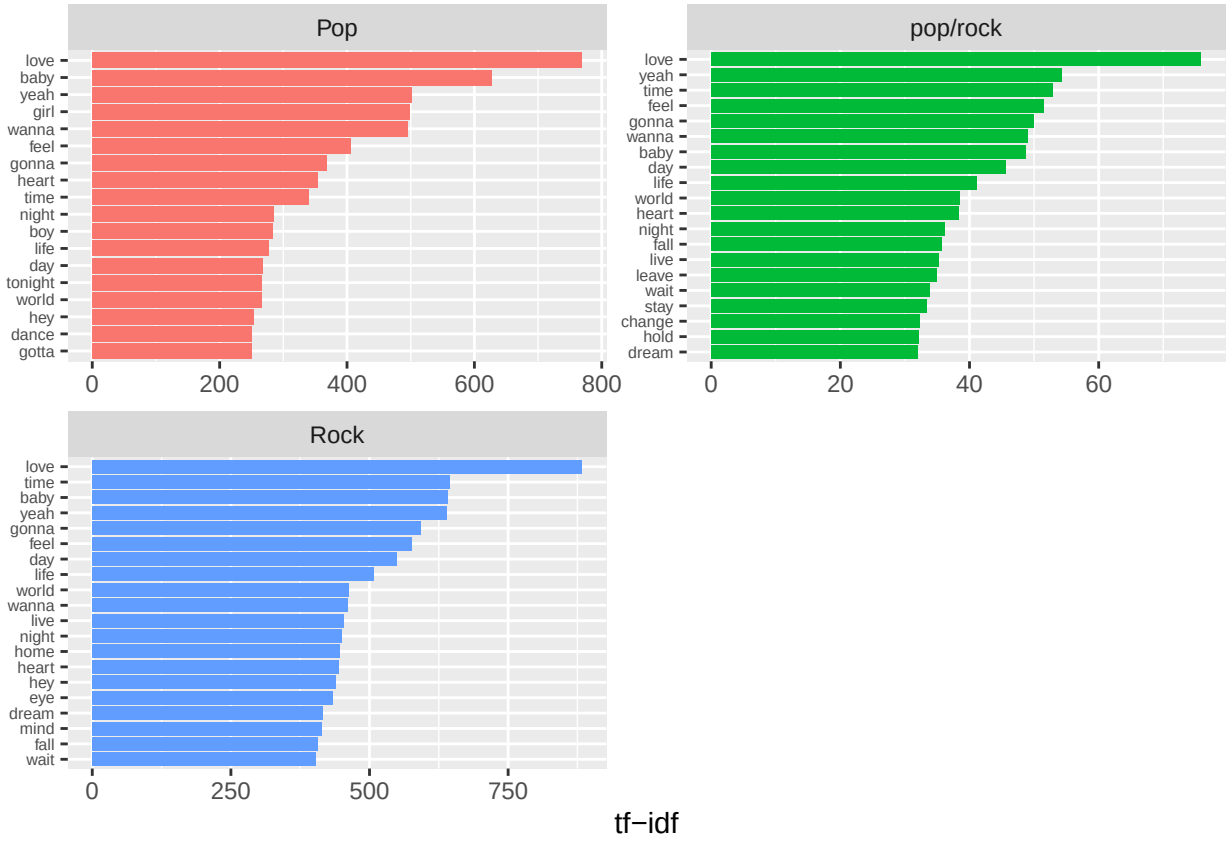
```
# TFIDF topwords
text_genre_tidy_rock= text_genre_tf_idf %>%
  filter(genre == "Rock")%>%
count(word, wt = tf_idf, sort = TRUE)%>%
  filter(!word == "chorus") %>% #remove
head(25)

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)%>%
  filter(!word == "chorus")%>%
  filter(!word == "ooh")%>% #remove
head(25)
```

We now plot the 20 most used words within each genre.

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



Rock wordcloud

EDA within the Rock genres.

```
text_tidy_rock = text_genre_tidy %>%
  filter(genre == "Rock")
```

```
library(wordcloud)
```

```
## Indlæser krævet pakke: RColorBrewer
```

```
text_tidy_rock %>%
count(word) %>%
with(wordcloud(word, n,
max.words = 50,
color = "blue"))
```

Pop wordcloud

EDA within the Pop genres.

```
text_tidy_Pop = text_genre_tidy %>%
filter(genre == "Pop")
```

```
text_tidy_Pop %>%
count(word) %>%
with(wordcloud(word, n,
max.words = 50,
color = "blue"))
```

Pop/Rock wordcloud

EDA within the Pop/Rock genres.

```
text_tidy_Pop_Rock = text_genre_tidy %>%
filter(genre == "pop/rock")
```

```
text_tidy_Pop_Rock %>%
count(word) %>%
with(wordcloud(word, n,
max.words = 50,
color = "blue"))
```

Sentiment Analysis

Rock_Pop

We do a sentiment analysis based on the Pop genre.

```
library(textdata)

text_tidy_Pop_Rock_index= text_tidy_Pop_Rock %>%
mutate(index= 1:n())
```

We use the lexicons "bing" and "afinn" to get a measure for positivity and negativity for each word. We use inner_join to only get the words we use from the lexicon.

```
#Bing
sentiment_bing <- text_tidy_Pop_Rock_index %>%
inner_join(get_sentiments("bing")) %>%
count(word, index = index %/% 100, sentiment) %>%
mutate(lexicon = 'Bing')
```

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

```
# Afinn
sentiment_afinn <- text_tidy_Pop_Rock_index %>%
inner_join(get_sentiments("afinn")) %>%
group_by(index = index %/% 100) %>%
summarise(sentiment = sum(value, na.rm = TRUE)) %>%
mutate(lexicon = 'AFINN')
```

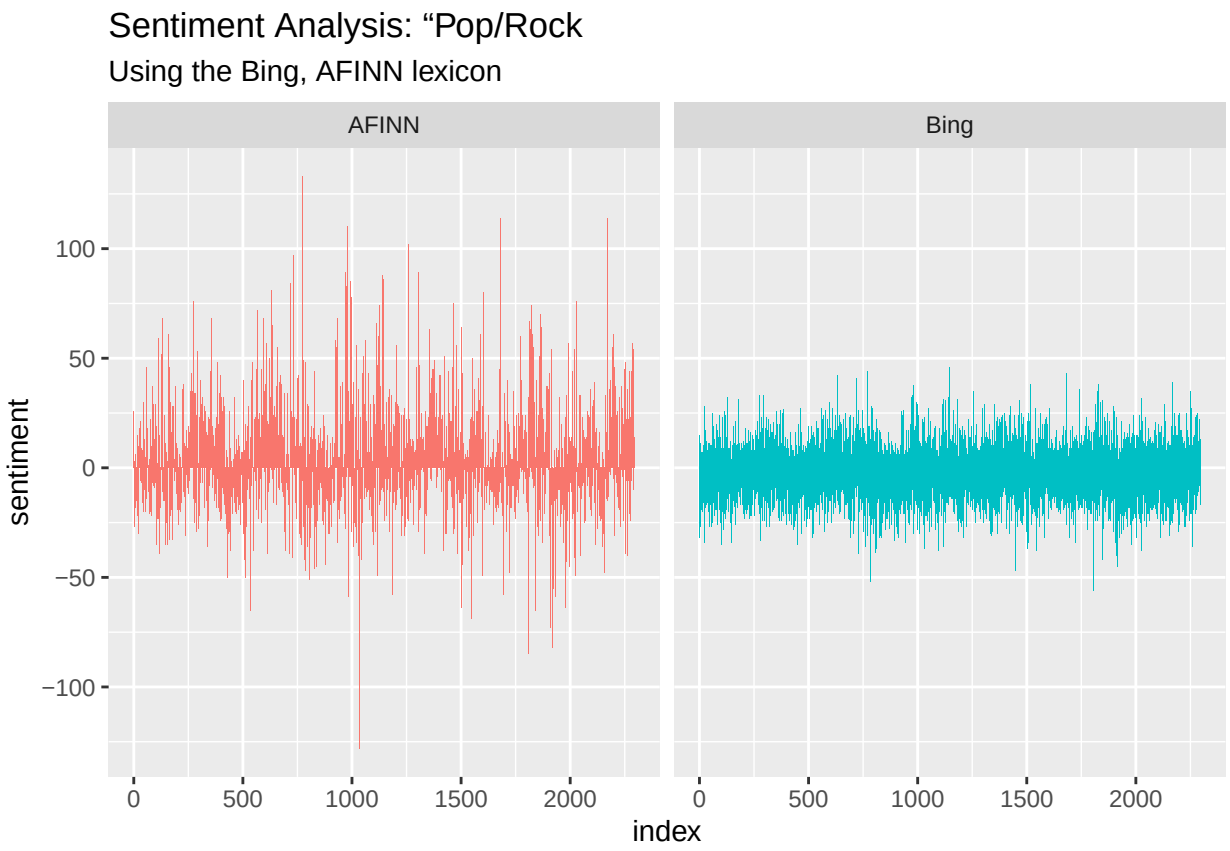
```
## Joining, by = "word"
```

We join the measures from both lexicons.

```
# Lets join them all together for plotting
sentiment_all <- sentiment_afinn %>%
bind_rows(sentiment_bing %>%
pivot_wider(names_from = sentiment, values_from = n, values_fill = 0) %>%
mutate(sentiment = positive - negative) %>%
select(index, sentiment, lexicon))
```

We create a plot for the distribution between negative and positive words within the Pop/Rock genre.

```
sentiment_all %>%
ggplot(aes(x = index, y = sentiment, fill = lexicon)) +
geom_col(show.legend = FALSE) +
facet_wrap(~ lexicon) +
labs(title = "Sentiment Analysis: "Pop/Rock",
subtitle = 'Using the Bing, AFINN lexicon')
```



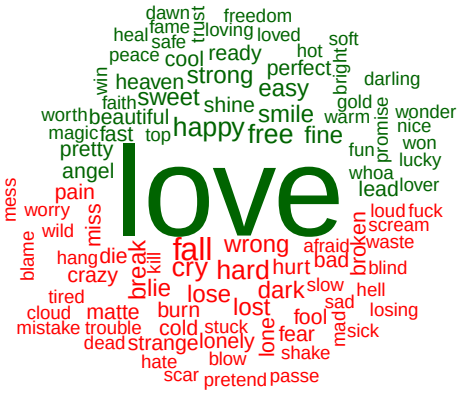
Senteminet wordcloud

We can now create a wordcloud looking at the positive and negative words in the Pop/Rock genre.

```
text_tidy_Pop_Rock %>%
inner_join(get_sentiments("bing")) %>%
count(word, sentiment, sort = TRUE) %>%
filter(sentiment %in% c("positive", "negative")) %>%
pivot_wider(names_from = sentiment, values_from = n, values_fill = 0) %>%
as.data.frame() %>%
remove_rownames() %>%
column_to_rownames("word") %>%
comparison.cloud(colors = c("darkgreen", "red"),
max.words = 100,
title.size = 1.5)
```

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

positive



negative

Pop

We do a sentiment analysis based on the Pop genre.

```
library(textdata)

text_tidy_Pop_index= text_tidy_Pop %>%
mutate(index= 1:n())
```

We use the lexicons “bing” and “afinn” to get a measure for positivity and negativity for each word. We use inner_join to only get the words we use from the lexicon.

```
#Bing
sentiment_bing_pop <- text_tidy_Pop_index %>%
inner_join(get_sentiments("bing")) %>%
count(word, index = index %/% 100, sentiment) %>%
mutate(lexicon = 'Bing')
```

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

```
# Afinn
sentiment_afinn_pop <- text_tidy_Pop_index %>%
inner_join(get_sentiments("afinn")) %>%
group_by(index = index %/% 100) %>%
summarise(sentiment = sum(value, na.rm = TRUE)) %>%
mutate(lexicon = 'AFINN')
```

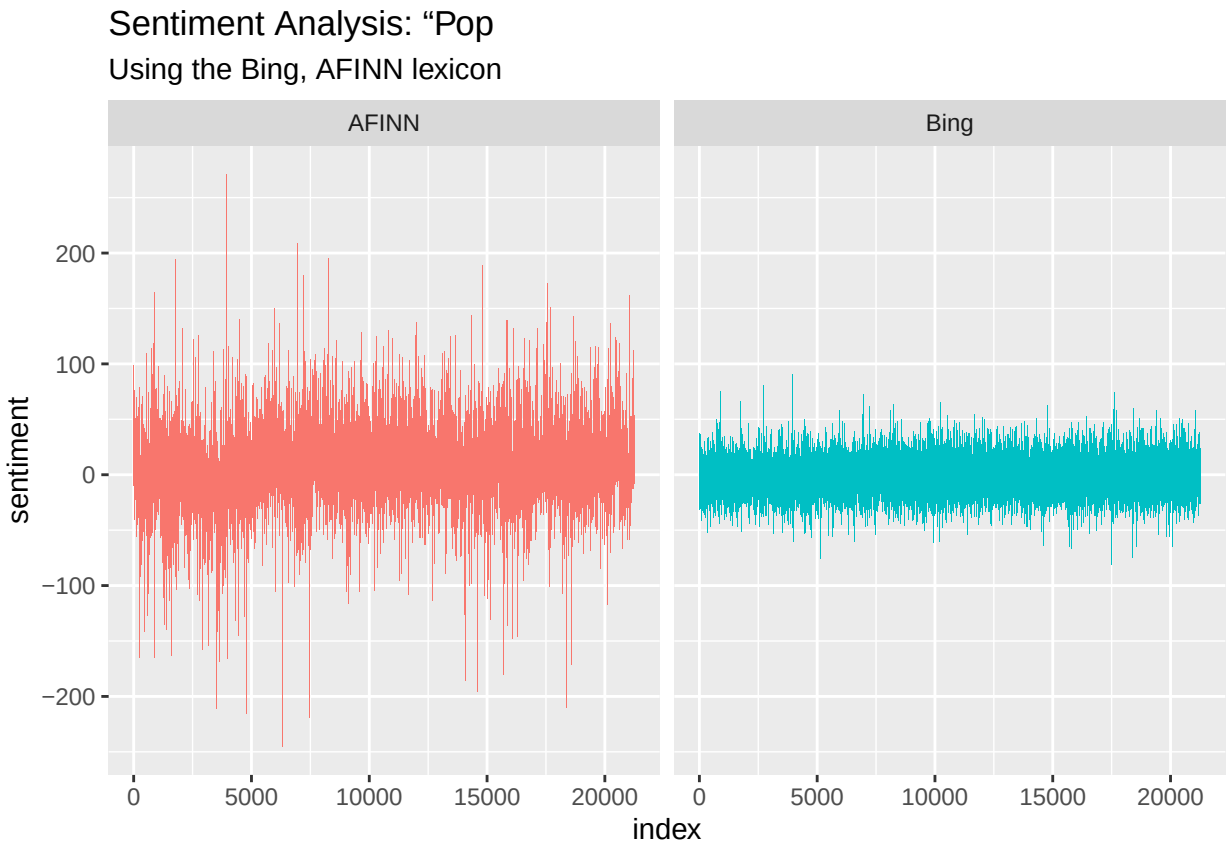
```
## Joining, by = "word"
```

We join the measures from both lexicons.

```
# Lets join them all together for plotting
sentiment_all_pop <- sentiment_afinn_pop %>%
bind_rows(sentiment_bing_pop %>%
pivot_wider(names_from = sentiment, values_from = n, values_fill = 0) %>%
mutate(sentiment = positive - negative) %>%
select(index, sentiment, lexicon))
```

We create a plot for the distribution between negative and positive words within the Pop genre.

```
sentiment_all_pop %>%
ggplot(aes(x = index, y = sentiment, fill = lexicon)) +
geom_col(show.legend = FALSE) +
facet_wrap(~ lexicon) +
labs(title = "Sentiment Analysis: "Pop",
subtitle = 'Using the Bing, AFINN lexicon')
```



Senteminet wordcloud

We can now create a wordcloud looking at the positive and negative words in the Pop genre.

```
text_tidy_Pop %>%
inner_join(get_sentiments("bing")) %>%
count(word, sentiment, sort = TRUE) %>%
filter(sentiment %in% c("positive", "negative")) %>%
pivot_wider(names_from = sentiment, values_from = n, values_fill = 0) %>%
as.data.frame() %>%
remove_rownames() %>%
column_to_rownames("word") %>%
comparison.cloud(colors = c("darkgreen", "red"),
max.words = 100,
title.size = 1.5)
```

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

positive

negative

Rock

We do a sentiment analysis based on the Rock genre.

```
library(textdata)

text_tidy_Rock_index= text_tidy_rock %>%
mutate(index= 1:n())
```

We use the lexicons “bing” and “afinn” to get a measure for positivity and negativity for each word. We use inner_join to only get the words we use from the lexicon.

```
#Bing
sentiment_bing_rock <- text_tidy_Rock_index %>%
inner_join(get_sentiments("bing")) %>%
count(word, index = index %/% 100, sentiment) %>%
mutate(lexicon = 'Bing')
```

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

```
# Afinn
sentiment_afinn_rock <- text_tidy_Rock_index %>%
inner_join(get_sentiments("afinn")) %>%
group_by(index = index %/% 100) %>%
```

```
summarise(sentiment = sum(value, na.rm = TRUE)) %>%
mutate(lexicon = 'AFINN')
```

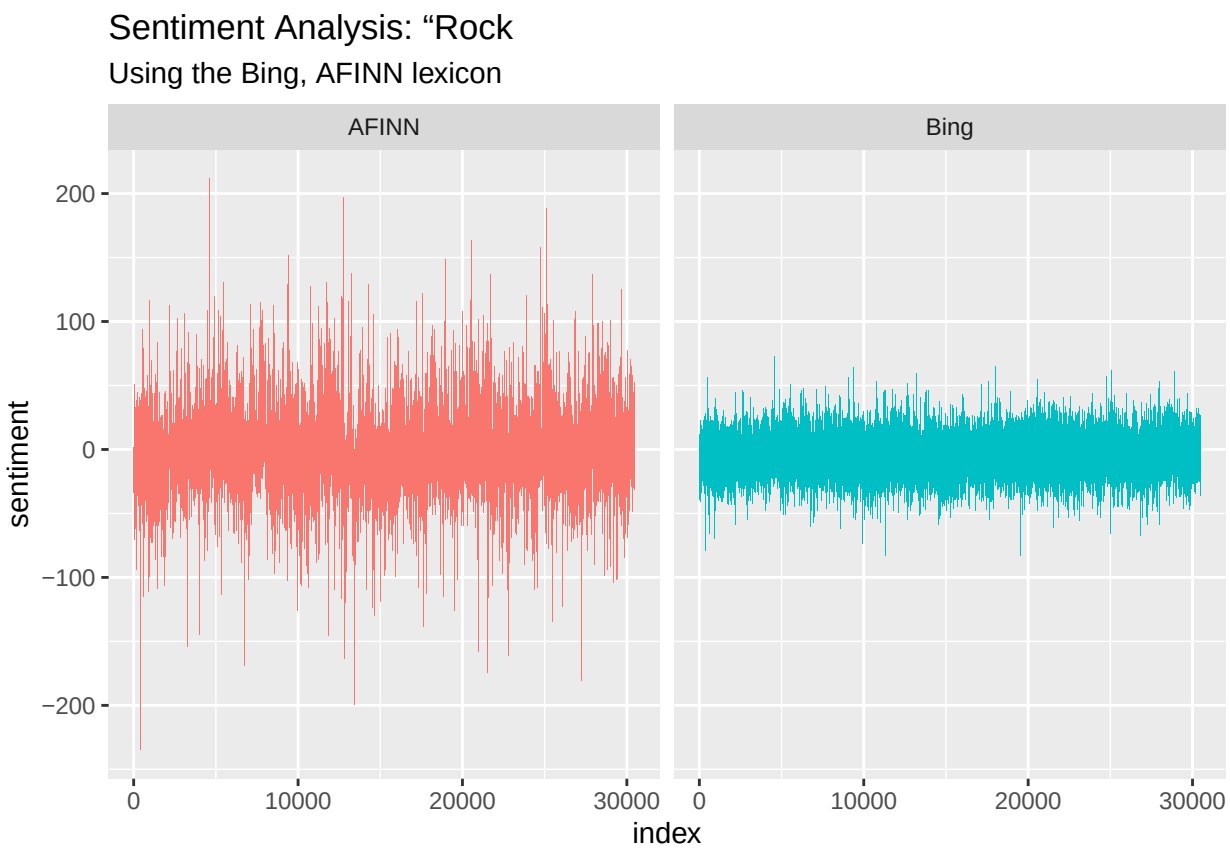
```
## Joining, by = "word"
```

We join the measures from both lexicons.

```
# Lets join them all together for plotting
sentiment_all_rock <- sentiment_afinn_rock %>%
bind_rows(sentiment_bing_rock %>%
pivot_wider(names_from = sentiment, values_from = n, values_fill = 0) %>%
mutate(sentiment = positive - negative) %>%
select(index, sentiment, lexicon))
```

We create a plot for the distribution between negative and positive words within the Rock genre.

```
sentiment_all_rock %>%
ggplot(aes(x = index, y = sentiment, fill = lexicon)) +
geom_col(show.legend = FALSE) +
facet_wrap(~ lexicon) +
labs(title = "Sentiment Analysis: "Rock",
subtitle = 'Using the Bing, AFINN lexicon')
```



Senteminet wordcloud

We can now create a wordcloud looking at the positive and negative words in the Rock genre.

```
text_tidy_rock %>%
inner_join(get_sentiments("bing")) %>%
count(word, sentiment, sort = TRUE) %>%
filter(sentiment %in% c("positive", "negative")) %>%
pivot_wider(names_from = sentiment, values_from = n, values_fill = 0) %>%
as.data.frame() %>%
remove_rownames() %>%
column_to_rownames("word") %>%
comparison.cloud(colors = c("darkgreen", "red"),
max.words = 100,
title.size = 1.5)
```

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

Bands analysis

We dont need the genre any more so we remove it.

```
data_band = data_start %>%
  filter(Idiom == "ENGLISH") %>%
  rename("Link" = "ALink") %>%
  inner_join(artists, by = c("Link")) %>%
  distinct() %>%
  rename(text=Lyric) %>%
  filter(Pop==1 | Rock==1) %>%
  select(Artist, text)
```

We want to see the most active artists.

```
data_band %>%
  count(Artist, sort = T)
```

```
## # A tibble: 974 x 2
##    Artist                n
##    <chr>             <int>
##  1 Elvis Presley       747
##  2 Glee                687
##  3 Chris Brown         562
##  4 Bee Gees            549
##  5 Bob Dylan           534
##  6 Neil Young          488
##  7 Van Morrison        485
##  8 Bruce Springsteen   477
##  9 Elvis Costello      474
## 10 Rod Stewart         435
## # ... with 964 more rows
```

We pick the top 3 artists (in our opinion) Green day, Bon Jovi and Red Hot Chili Peppers!

```
best_artists= data_band %>%
  filter(Artist %in% c("Green Day",  "Bon Jovi" , "Red Hot Chili Peppers" ))
```

First we tokenize the data.

```
text_band_tidy = best_artists %>% unnest_tokens(word, text, token = "words")

head(text_band_tidy)
```

```
## # A tibble: 6 x 2
##   Artist   word
##   <chr>    <chr>
## 1 Bon Jovi this
## 2 Bon Jovi romeo
## 3 Bon Jovi is
## 4 Bon Jovi bleeding
## 5 Bon Jovi but
## 6 Bon Jovi you
```

We remove short words and stopwords.

```
text_band_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.

```
text_band_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).

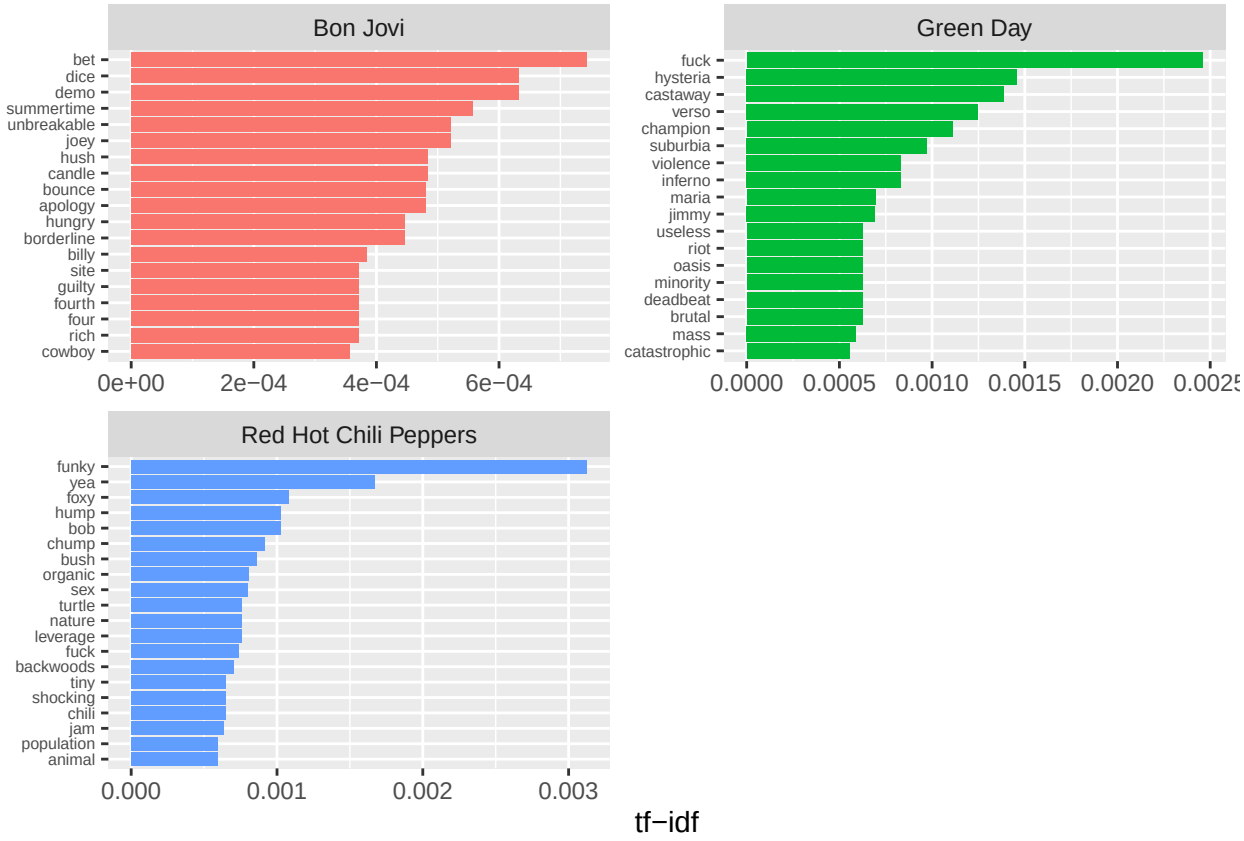
```
# TFIDF weights
text_band_tf_idf= text_band_tidy %>%
group_by(Artist) %>%
  count(word, sort = TRUE) %>%
  ungroup() %>%
  bind_tf_idf(word, Artist, n) %>%
  arrange(desc(tf_idf))
```

We show the 25 most common words within the 3 artists.

```
# TFIDF topwords
text_band_tidy_Bon_Jovi= text_band_tf_idf %>%
  filter(Artist == "Bon Jovi")%>%
count(word, wt = tf_idf, sort = TRUE)%>%
  filter(!word == "mo") %>% #remove
head(25)

text_band_tidy_Green_Day= text_band_tf_idf %>%
  filter(Artist == "Green Day")%>%
count(word, wt = tf_idf, sort = TRUE)%>%
  filter(!word == "intro")%>%
  filter(!word == "riff") %>% #remove
head(25)

text_band_tidy_RHCP= text_band_tf_idf %>%
  filter(Artist == "Red Hot Chili Peppers")%>%
count(word, wt = tf_idf, sort = TRUE)%>%
  filter(!word == "co")%>%
  filter(!word == "cos") %>% #remove
head(25)
```

We now plot the 20 most used words within each genre.

```
labels_words_band <- text_band_tf_idf %>%
  group_by(Artist) %>%
  count(word, wt = tf_idf, sort = TRUE, name = "tf_idf") %>%
  dplyr::slice(1:20)%>%
  filter(!word == "mo")%>%
  filter(!word == "intro")%>%
  filter(!word == "riff")%>%
  filter(!word == "co")%>%
  filter(!word == "cos") %>%
  ungroup()

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



Songs of the top 3 artist

```
# Greenday

sentiment_green_day= text_band_tidy %>%
  filter(Artist == "Green Day") %>%
  inner_join(get_sentiments("bing")) %>% # pull out only sentiment words
  count(sentiment) %>% # count the # of positive & negative words
  spread(sentiment, n, fill = 0) %>% # made data wide rather than narrow
  mutate(sentiment = positive - negative) # # of positive words - # of negative owrds
```

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

```
sentiment_green_day
```

```
## # A tibble: 1 x 3
##   negative positive sentiment
##      <dbl>    <dbl>     <dbl>
## 1     3309      923     -2386
```

```
# Bon Jovi

sentiment_Bon_Jovi= text_band_tidy %>%
  filter(Artist == "Bon Jovi") %>%
  inner_join(get_sentiments("bing")) %>% # pull out only sentiment words
  count(sentiment) %>% # count the # of positive & negative words
  spread(sentiment, n, fill = 0) %>% # made data wide rather than narrow
  mutate(sentiment = positive - negative) # # of positive words - # of negative owrds
```

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

```
sentiment_Bon_Jovi
```

```
## # A tibble: 1 x 3
##   negative positive sentiment
##      <dbl>    <dbl>     <dbl>
## 1     3819     2445     -1374
```

```
# Red Hot Chilie Pepper

sentiment_RHCP= text_band_tidy %>%
  filter(Artist == "Red Hot Chili Peppers") %>%
  inner_join(get_sentiments("bing")) %>% # pull out only sentiment words
  count(sentiment) %>% # count the # of positive & negative words
  spread(sentiment, n, fill = 0) %>% # made data wide rather than narrow
  mutate(sentiment = positive - negative) # # of positive words - # of negative owrds
```

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

```
sentiment_RHCP
```

```
## # A tibble: 1 x 3
##   negative positive sentiment
##      <dbl>    <dbl>     <dbl>
## 1     2686     1996      -690
```

We can see all the artist use more negative laden words than positive

To see sentiment for each song we can join the song names again.

```
data_song_name = data_start %>%
  filter(Idiom == "ENGLISH") %>%
  rename("Link" = "ALink") %>%
  inner_join(artists, by = c("Link")) %>%
  distinct() %>%
  filter(Artist %in% c("Bon Jovi", "Green Day", "Red Hot Chili Peppers")) %>%
  rename(text=Lyric) %>%
  filter(Pop==1 | Rock==1) %>%
  select(SName, text)
```

We tokenize

```
text_song_tidy = data_song_name %>% unnest_tokens(word, text, token = "words")
```

We remove short words and stopwords.

```
text_song_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.

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

We will now wheight by tf-idf

```
# TFIDF weights
text_song_tf_idf= text_song_tidy %>%
group_by(SName) %>%
  count(word, sort = TRUE) %>%
  ungroup() %>%
  bind_tf_idf(word, SName, n) %>%
  arrange(desc(tf_idf))
```

We can now add the band name for our chossen artists

```
data_song_artist = data_start %>%
  filter(Idiom == "ENGLISH") %>%
  rename("Link" = "ALink") %>%
  inner_join(artists, by = c("Link")) %>%
  distinct() %>%
  filter(Artist %in% c("Bon Jovi", "Green Day", "Red Hot Chili Peppers")) %>%
  rename(text=Lyric) %>%
  filter(Pop==1 | Rock==1) %>%
  select(Artist,SName)


text_song_tf_idf %<>%
  inner_join(data_song_artist, by= c("SName"))

# For Green Day

green_day_songs=text_song_tf_idf %>%
  filter(Artist == "Green Day") %>%
  arrange(desc(tf_idf))%>%
  inner_join(get_sentiments("bing"))%>%
  mutate(sentiment= ifelse(sentiment == "negative", -1, 1)) %>%
  group_by(SName) %>%
  summarise(sum= sum(sentiment)) %>%
  mutate(sentiment_song= ifelse(sum > 0, "positive", ifelse(sum == 0, "neutral", "negative")))%>%
  count(sentiment_song)
```

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

```
# For RHCP

RHCP_songs=text_song_tf_idf %>%
  filter(Artist == "Red Hot Chili Peppers") %>%
  arrange(desc(tf_idf))%>%
  inner_join(get_sentiments("bing"))%>%
  mutate(sentiment= ifelse(sentiment == "negative", -1, 1)) %>%
  group_by(SName) %>%
  summarise(sum= sum(sentiment)) %>%
  mutate(sentiment_song= ifelse(sum > 0, "positive", ifelse(sum == 0, "neutral", "negative")))%>%
  count(sentiment_song)
```

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

```
# Bon Jovi

Bon_Jovi_songs=text_song_tf_idf %>%
  filter(Artist == "Bon Jovi") %>%
  arrange(desc(tf_idf))%>%
  inner_join(get_sentiments("bing"))%>%
  mutate(sentiment= ifelse(sentiment == "negative", -1, 1)) %>%
  group_by(SName) %>%
```

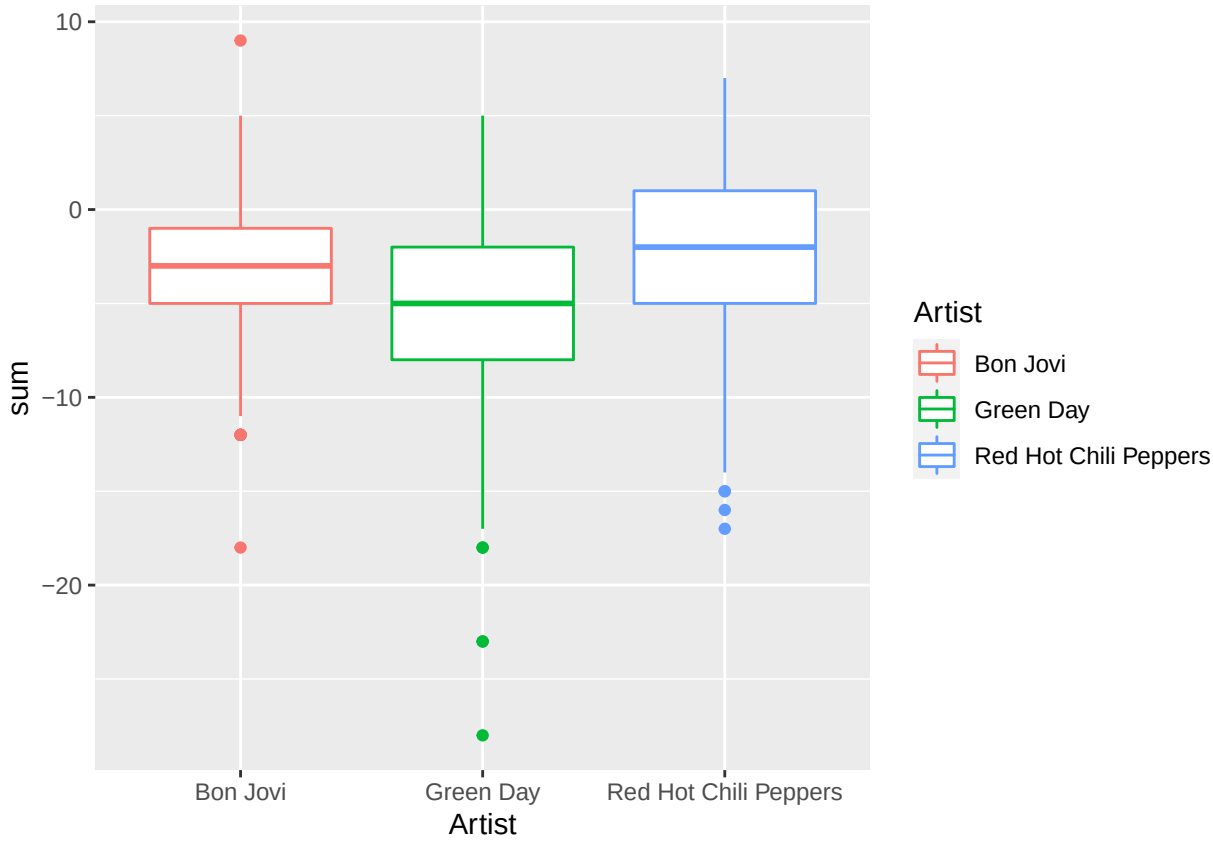
```
  summarise(sum= sum(sentiment)) %>%
  mutate(sentiment_song= ifelse(sum > 0, "positive", ifelse(sum == 0, "neutral", "negative")))%>%
  count(sentiment_song)
```

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

```
# For all

all_songs=text_song_tf_idf %>%
  arrange(desc(tf_idf))%>%
  inner_join(get_sentiments("bing"))%>%
  mutate(sentiment= ifelse(sentiment == "negative", -1, 1)) %>%
  group_by(SName) %>%
  summarise(sum= sum(sentiment)) %>%
  mutate(sentiment_song= ifelse(sum > 0, "positive", ifelse(sum == 0, "neutral", "negative")))%>%
  inner_join(data_song_artist, by= c("SName"))
```

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

```
ggplot(all_songs, aes(x = Artist, y = sum, color = Artist)) +
  geom_boxplot()
```



We can here that the overall score of the songs seems to be negative for all three artists. We can tho see that RHCP on the averrage has ths most positive songs looking at the three artists.

Sentiment over time

We found a dataset including release date for songs on spotify

```
data_releaseyear <- read_csv("data.csv") # ligger på Github
```

```
## Rows: 169909 Columns: 19
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (4): artists, id, name, release_date
## dbl (15): acousticness, danceability, duration_ms, energy, explicit, instrum...
```

```
##
## 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.
```

```
release_year_bon_jovi= data_releaseyear %>%
  filter(artists == "['Bon Jovi']") %>%
  select(name, year)

release_year_RHCP= data_releaseyear %>%
  filter(artists == "['Red Hot Chili Peppers']") %>%
  select(name, year)

release_year_Green_Day= data_releaseyear %>%
  filter(artists == "['Green Day']") %>%
  select(name, year)
```

We will innerjoin with the datasets above

```
#Bon Jovi

Bon_Jovi_songs=text_song_tf_idf %>%
  filter(Artist == "Bon Jovi") %>%
  arrange(desc(tf_idf))%>%
  inner_join(get_sentiments("bing"))%>%
  mutate(sentiment= ifelse(sentiment == "negative", -1, 1)) %>%
  group_by(SName) %>%
  summarise(sum= sum(sentiment)) %>%
  mutate(sentiment_song= ifelse(sum > 0, "positive", ifelse(sum == 0, "neutral", "negative"))) %>%
  inner_join(release_year_bon_jovi, by= c("SName" = "name")) %>%
  distinct(SName, .keep_all = T)
```

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

```
#RHCP
RHCP_songs=text_song_tf_idf %>%
  filter(Artist == "Red Hot Chili Peppers") %>%
  arrange(desc(tf_idf))%>%
  inner_join(get_sentiments("bing"))%>%
```

```
  mutate(sentiment= ifelse(sentiment == "negative", -1, 1)) %>%
  group_by(SName) %>%
  summarise(sum= sum(sentiment)) %>%
  mutate(sentiment_song= ifelse(sum > 0, "positive", ifelse(sum == 0, "neutral", "negative"))) %>%
  inner_join(release_year_RHCP, by= c("SName" = "name")) %>%
  distinct(SName, .keep_all = T)
```

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

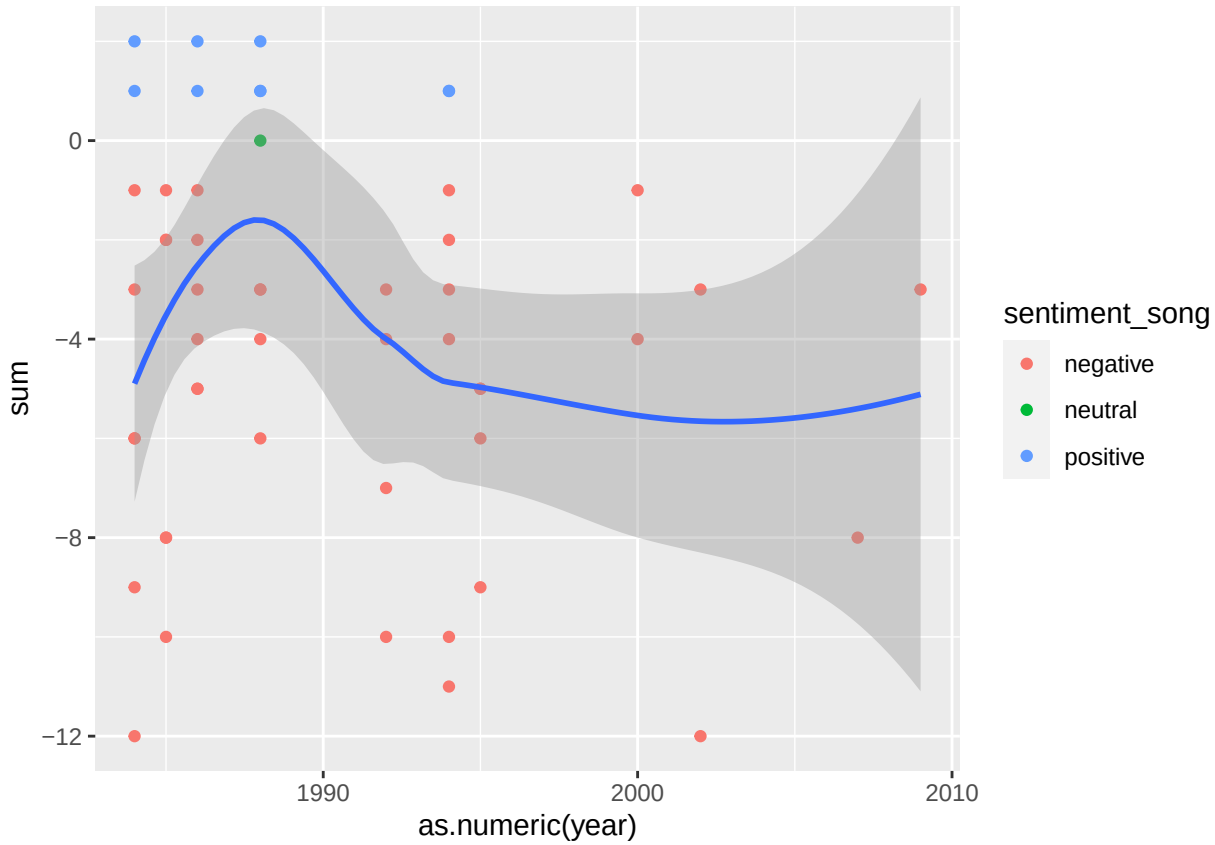
```
## Green Day

Green_day_songs=text_song_tf_idf %>%
  filter(Artist == "Green Day") %>%
  arrange(desc(tf_idf))%>%
  inner_join(get_sentiments("bing"))%>%
  mutate(sentiment= ifelse(sentiment == "negative", -1, 1)) %>%
  group_by(SName) %>%
  summarise(sum= sum(sentiment)) %>%
  mutate(sentiment_song= ifelse(sum > 0, "positive", ifelse(sum == 0, "neutral", "negative"))) %>%
  inner_join(release_year_Green_Day, by= c("SName" = "name")) %>%
  distinct(SName, .keep_all = T)
```

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

Development over time

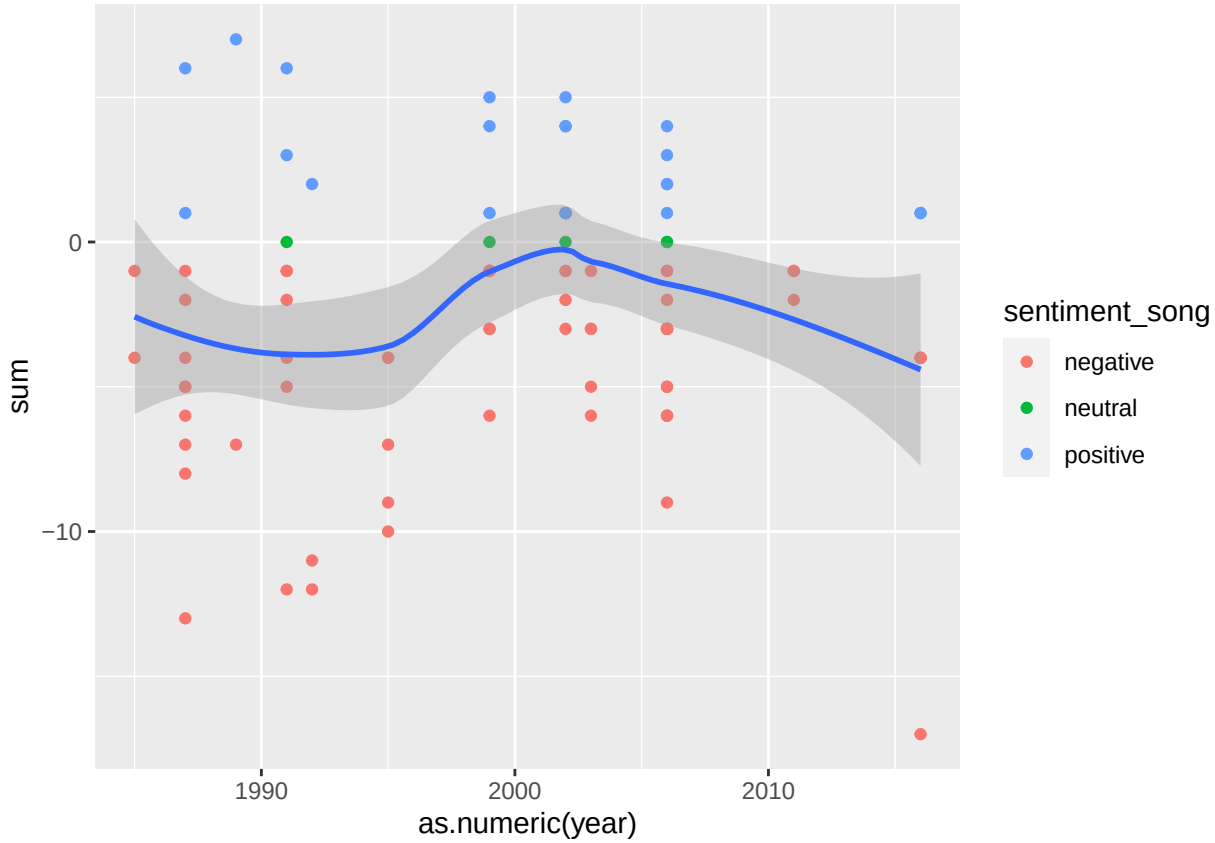
```
## Bon Jovi
ggplot(Bon_Jovi_songs, aes(x = as.numeric(year), y = sum)) +
  geom_point(aes(color = sentiment_song))+ # add points to our plot, color-coded by president
  geom_smooth(method = "auto") # pick a method & fit a model
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



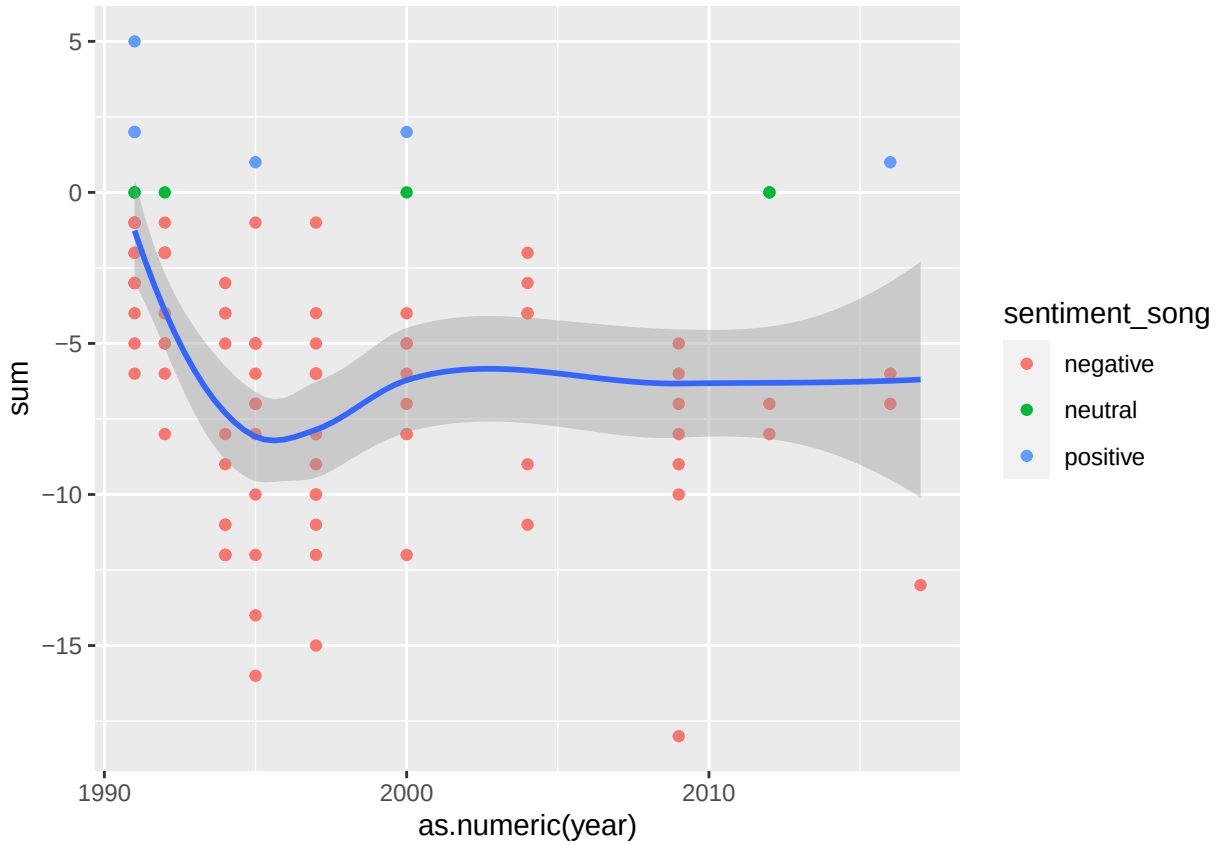

```
## RHCP

ggplot(RHCP_songs, aes(x = as.numeric(year), y = sum)) +
  geom_point(aes(color = sentiment_song))+ # add points to our plot, color-coded by president
  geom_smooth(method = "auto") # pick a method & fit a model
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
## Green Day

ggplot(Green_day_songs, aes(x = as.numeric(year), y = sum)) +
  geom_point(aes(color = sentiment_song))+ # add points to our plot, color-coded by president
  geom_smooth(method = "auto") # pick a method & fit a model
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



We can now see the development of the sentiment of the songs from the three artists. it looks like the artists at some point go for more positive songs, but return to more negative again.

Saddness og joy

We will now look at the development of Saddness or joy in the songs of the three artists

```
#NRC
sentiment_bing <- text_tidy_Pop_Rock_index %>%
inner_join(get_sentiments("nrc")) %>%
count(word, index = index %/% 100, sentiment) %>%
mutate(lexicon = 'Bing')
```

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

```
#Bon Jovi

Bon_Jovi_songs_joy_sadness=text_song_tf_idf %>%
  filter(Artist == "Bon Jovi") %>%
  arrange(desc(tf_idf))%>%
  inner_join(get_sentiments("nrc"))%>%
  filter(sentiment %in% c("sadness", "joy")) %>%
  mutate(sentiment= ifelse(sentiment == "sadness", -1, 1)) %>%
  group_by(SName) %>%
```

```
  summarise(sum= sum(sentiment)) %>%
  mutate(sentiment_song= ifelse(sum > 0, "joy", ifelse(sum == 0, "neutral", "sadness"))) %>%
  inner_join(release_year_bon_jovi, by= c("SName" = "name")) %>%
  distinct(SName, .keep_all = T)
```

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

```
## Green Day

Green_Day_songs_joy_sadness=text_song_tf_idf %>%
  filter(Artist == "Green Day") %>%
  arrange(desc(tf_idf))%>%
  inner_join(get_sentiments("nrc"))%>%
  filter(sentiment %in% c("sadness", "joy")) %>%
  mutate(sentiment= ifelse(sentiment == "sadness", -1, 1)) %>%
  group_by(SName) %>%
  summarise(sum= sum(sentiment)) %>%
  mutate(sentiment_song= ifelse(sum > 0, "joy", ifelse(sum == 0, "neutral", "sadness"))) %>%
  inner_join(release_year_Green_Day, by= c("SName" = "name")) %>%
  distinct(SName, .keep_all = T)
```

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

```
## RHCP

RHCP_songs_joy_sadness=text_song_tf_idf %>%
  filter(Artist == "Red Hot Chili Peppers") %>%
  arrange(desc(tf_idf))%>%
  inner_join(get_sentiments("nrc"))%>%
  filter(sentiment %in% c("sadness", "joy")) %>%
  mutate(sentiment= ifelse(sentiment == "sadness", -1, 1)) %>%
  group_by(SName) %>%
  summarise(sum= sum(sentiment)) %>%
  mutate(sentiment_song= ifelse(sum > 0, "joy", ifelse(sum == 0, "neutral", "sadness"))) %>%
  inner_join(release_year_RHCP, by= c("SName" = "name")) %>%
  distinct(SName, .keep_all = T)
```

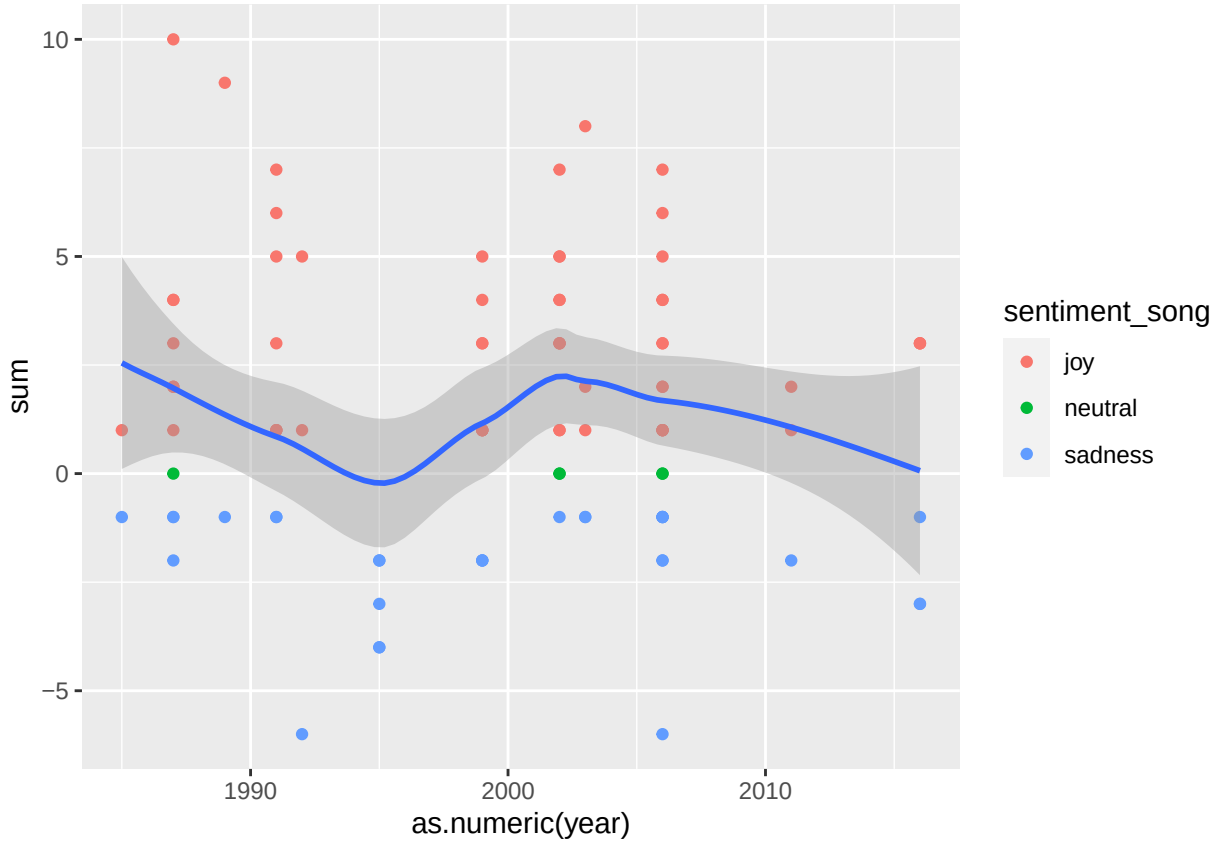
```
## Joining, by = "word"
```

```
## Bon Jovi
ggplot(Bon_Jovi_songs_joy_sadness, aes(x = as.numeric(year), y = sum)) +
  geom_point(aes(color = sentiment_song))+ # add points to our plot, color-coded by president
  geom_smooth(method = "auto") # pick a method & fit a model
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

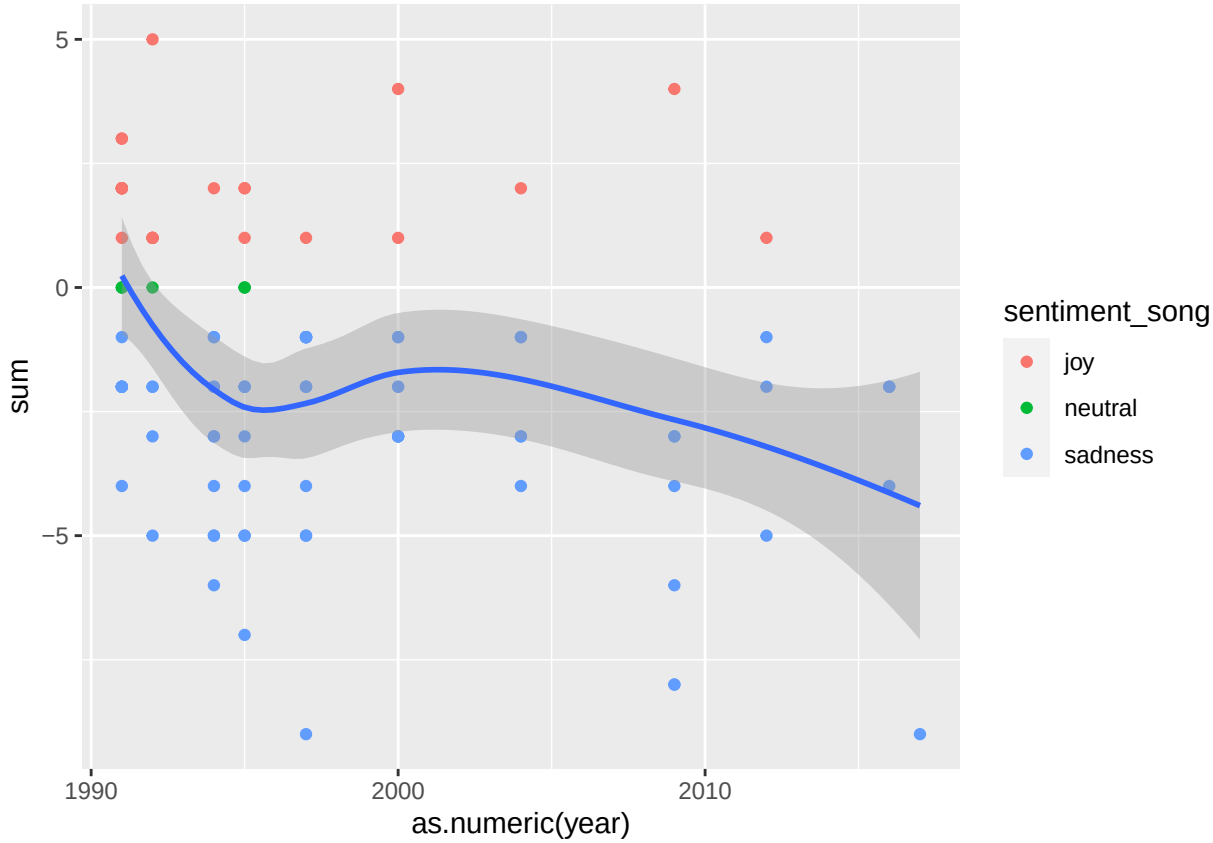

```
## RHCP

ggplot(RHCP_songs_joy_sadness, aes(x = as.numeric(year), y = sum)) +
  geom_point(aes(color = sentiment_song))+ # add points to our plot, color-coded by president
  geom_smooth(method = "auto") # pick a method & fit a model
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
## Green Day

ggplot(Green_Day_songs_joy_sadness, aes(x = as.numeric(year), y = sum)) +
  geom_point(aes(color = sentiment_song))+ # add points to our plot, color-coded by president
  geom_smooth(method = "auto") # pick a method & fit a model
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



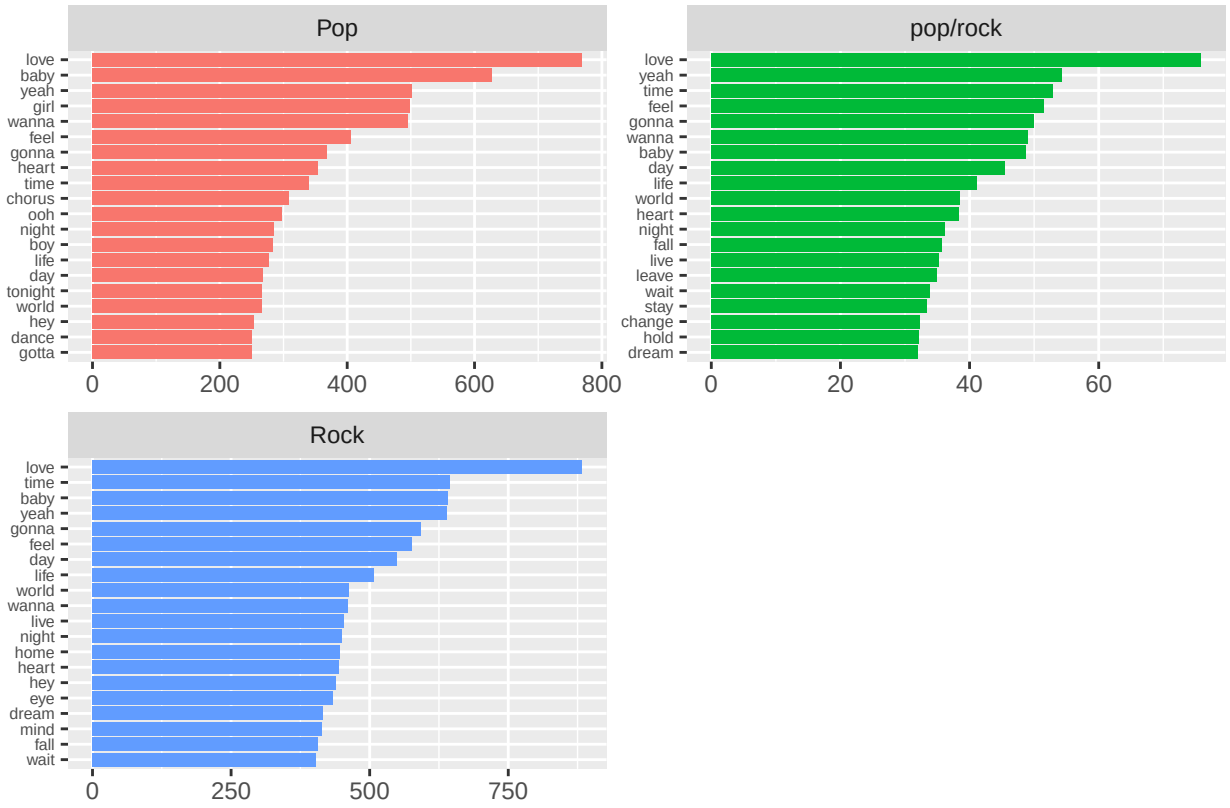
It looks like the artists song tend to be more sadd over time, it's actually kinda of sad to see that.....

Neural Network Bing

Labels

```
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"
```

Using our bing data set

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

The models

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'
```

```
## Det følgende objekt er maskeret fra 'package:textdata':
##
##     dataset_imdb
```

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

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

```
#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 :)

```
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

```
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 1 1 1 0 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
## _________________________________________________________________________
```

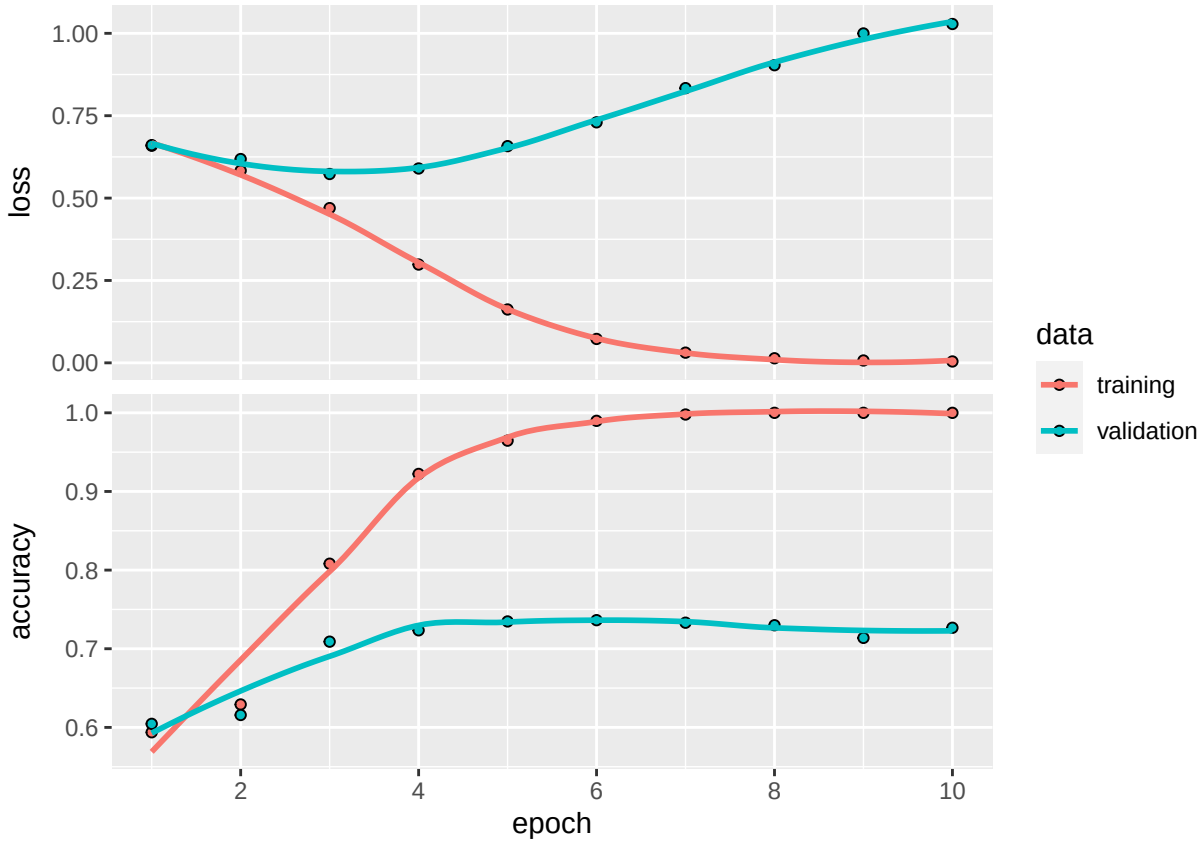
And now the model is run 10 times with a batch size of 256

```
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
## 1.7061692 0.5903614
```

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
## _________________________________________________________________________
```

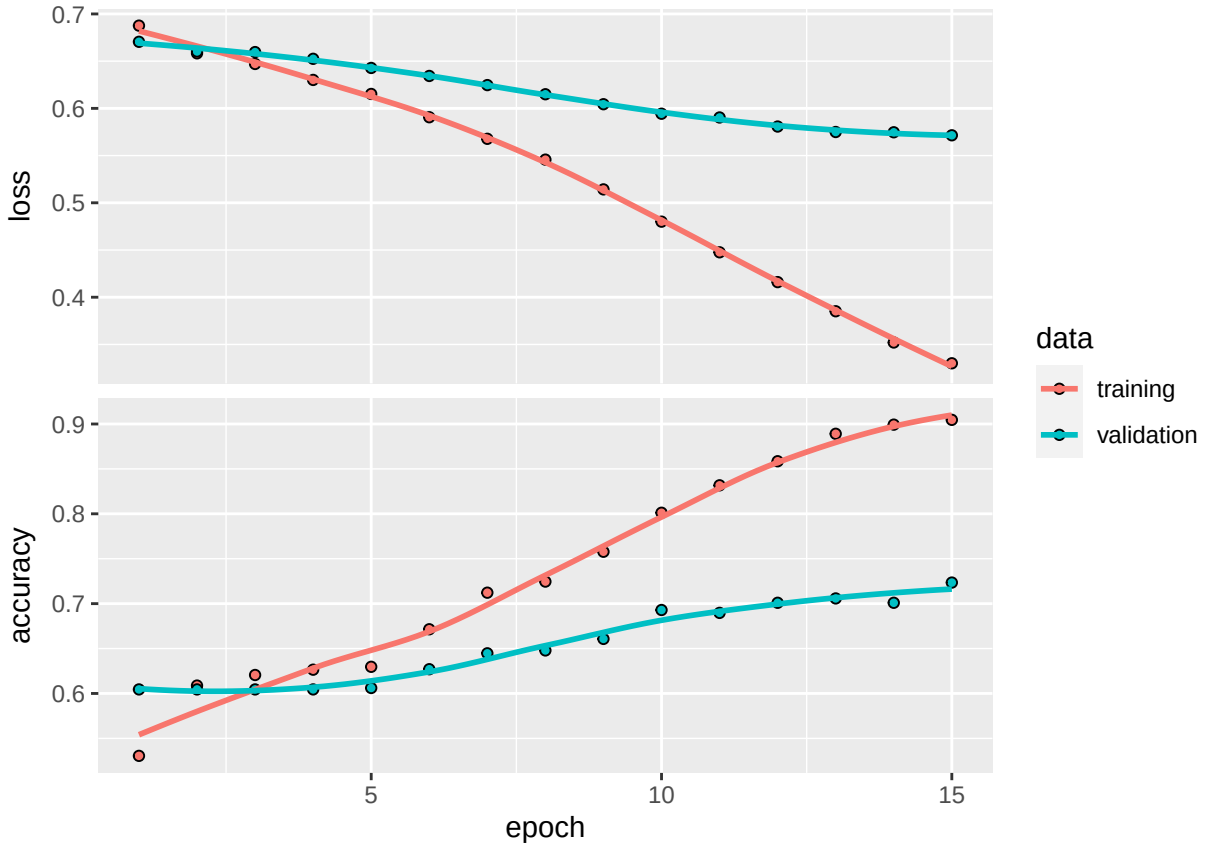
And now the model is run 10 times with a batch size of 512, so a bigger batch size than the baseline model.

```
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
## 0.6898360 0.6301205
```

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 0 0 0 0 0 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.

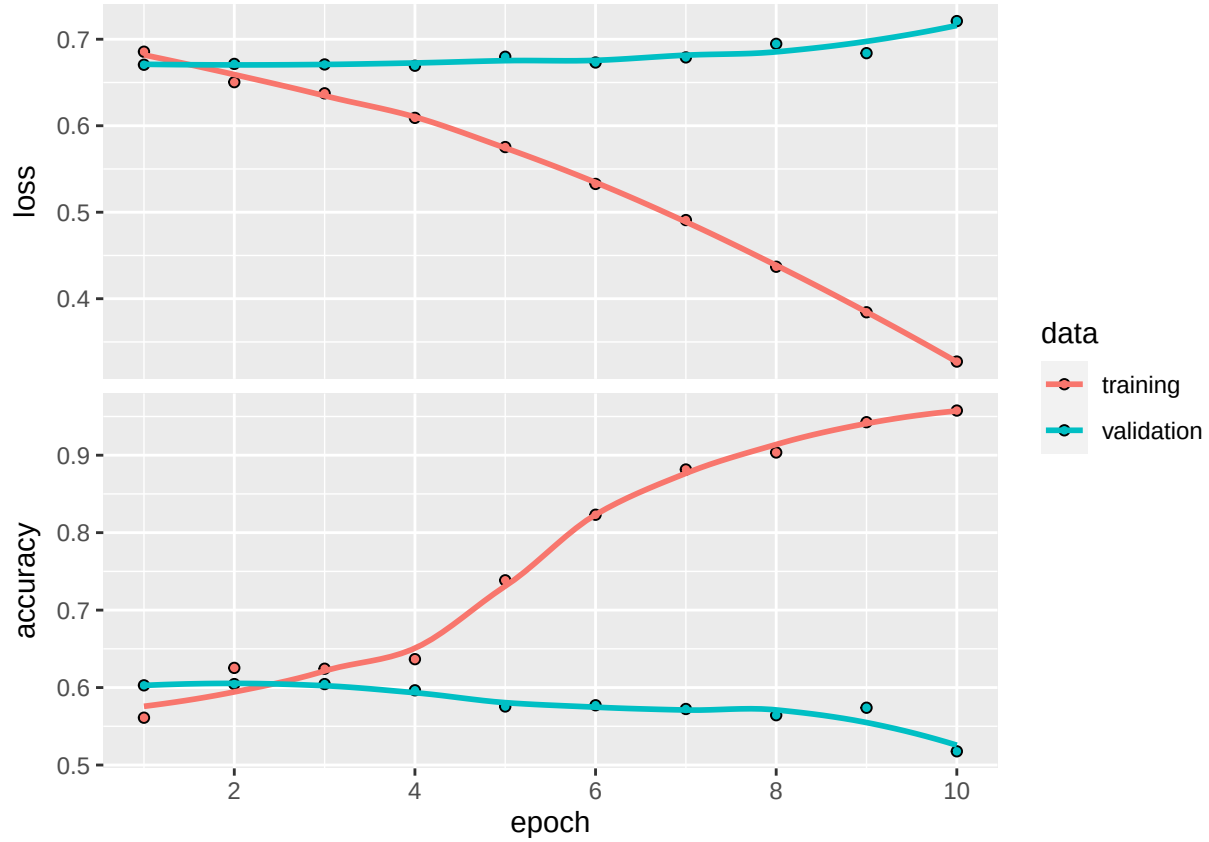
```
model_rnn %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = "accuracy"
)
```

And run our model

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

```
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
## 0.7206326 0.5361446
```

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.

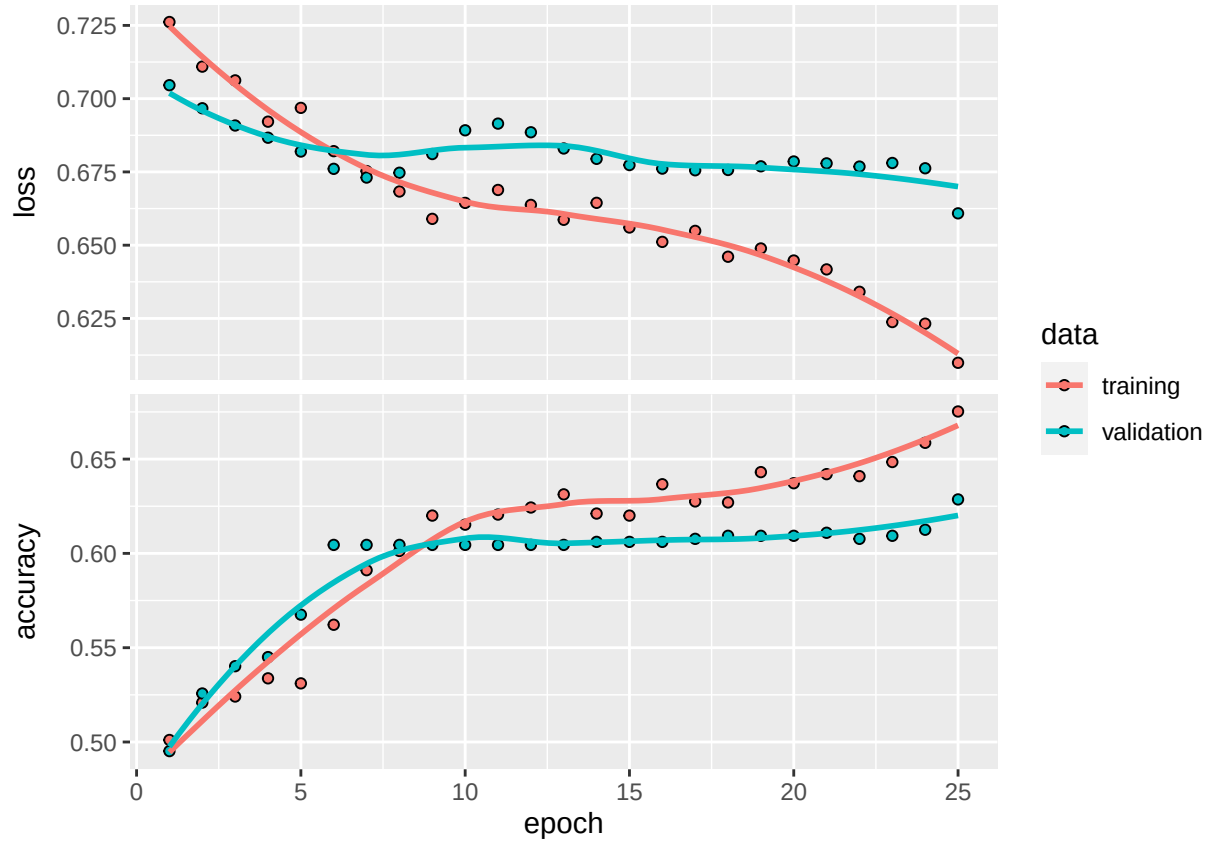
```
model_rnn2 %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = "accuracy"
)
```

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
## 0.7089092 0.5493976
```

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)
```

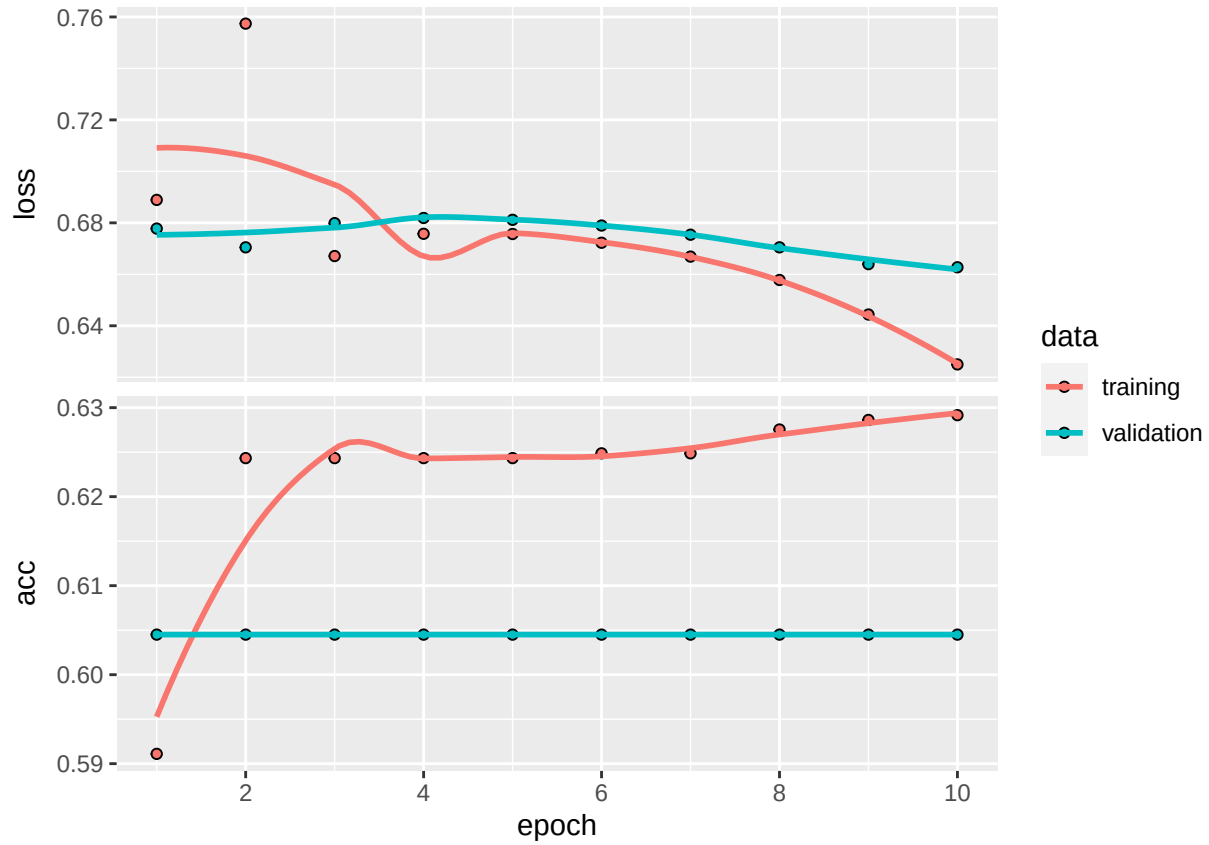
```
## Model: "sequential_4"
## _________________________________________________________________________________
## Layer (type)                         Output Shape                     Param #
## =================================================================================
## embedding_2 (Embedding)              (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
## 0.6596083 0.6361446
```

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 == negative, "negative", ifelse(label == joy, "joy","none lab
  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, "sadness", ifelse(label == joy, "joy","none label
  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, "trust"= 3)) %>%
  as.matrix()


y_test_data5= test_data5 %>% select('y') %>%
  mutate(y= recode(y, "joy" = 0, "sadness" = 1,"fear" =2, "trust"= 3)) %>%
  as.matrix()
```

```
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)
```

```
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)
```

```
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.

```
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.

```
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
## _________________________________________________________________________
```

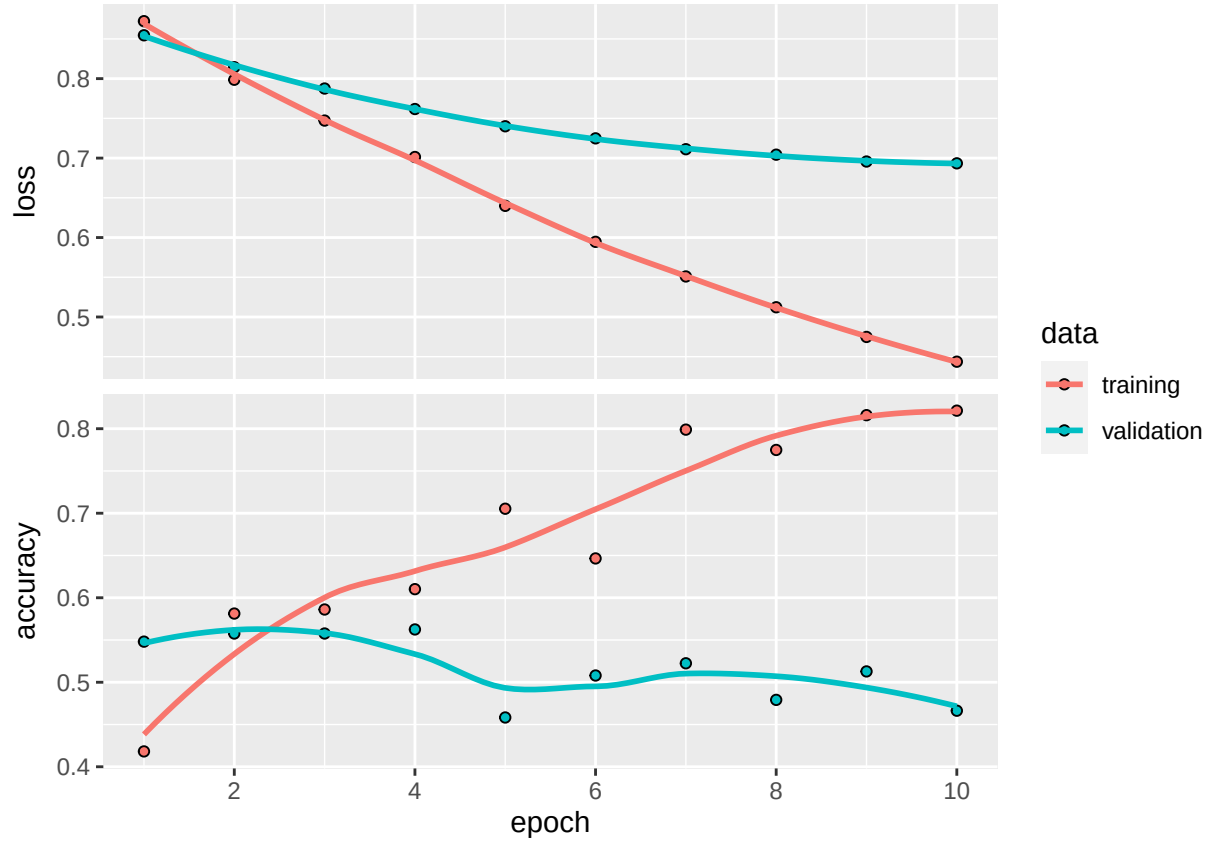
And now the model is run 10 times with a batch size of 256

```
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'
```



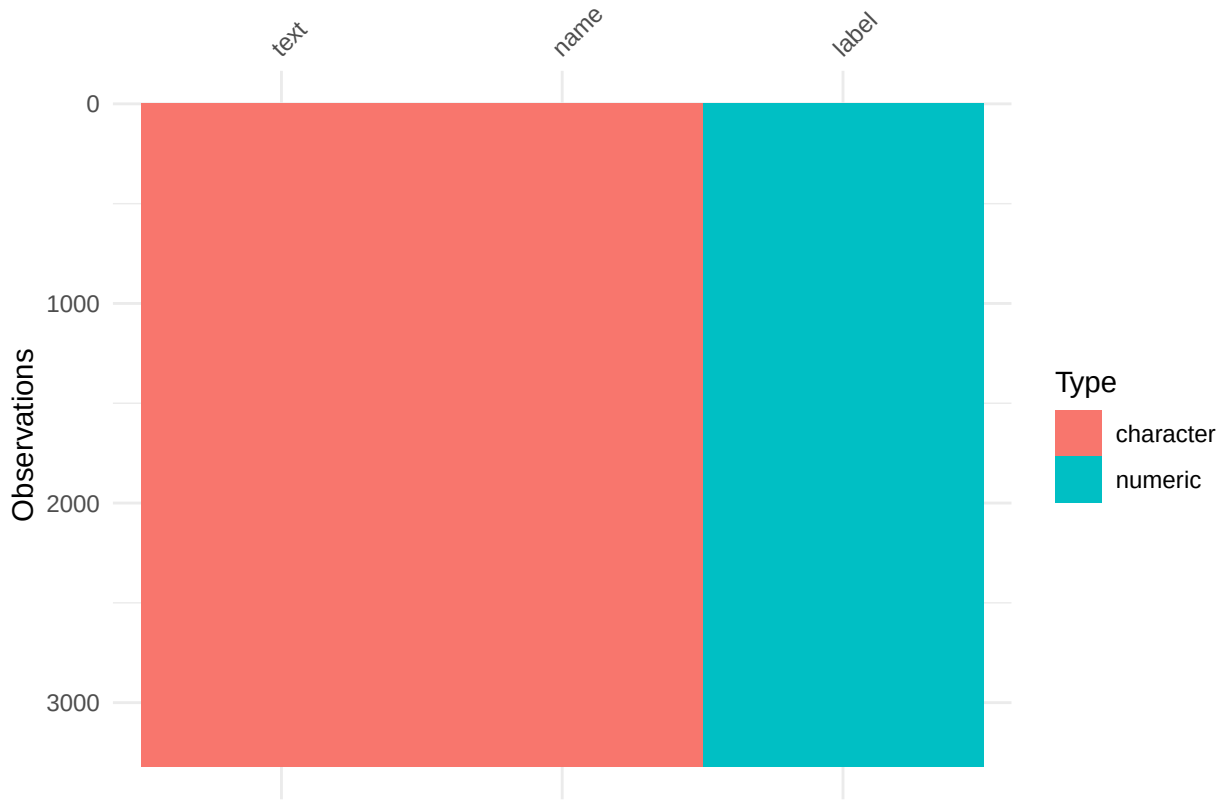

```
metrics10 = model5 %>% evaluate(x_test10, one_hot_test_labels); metrics10
```

```
##      loss  accuracy
## 0.7462488 0.3846154
```

Machine learning Binary

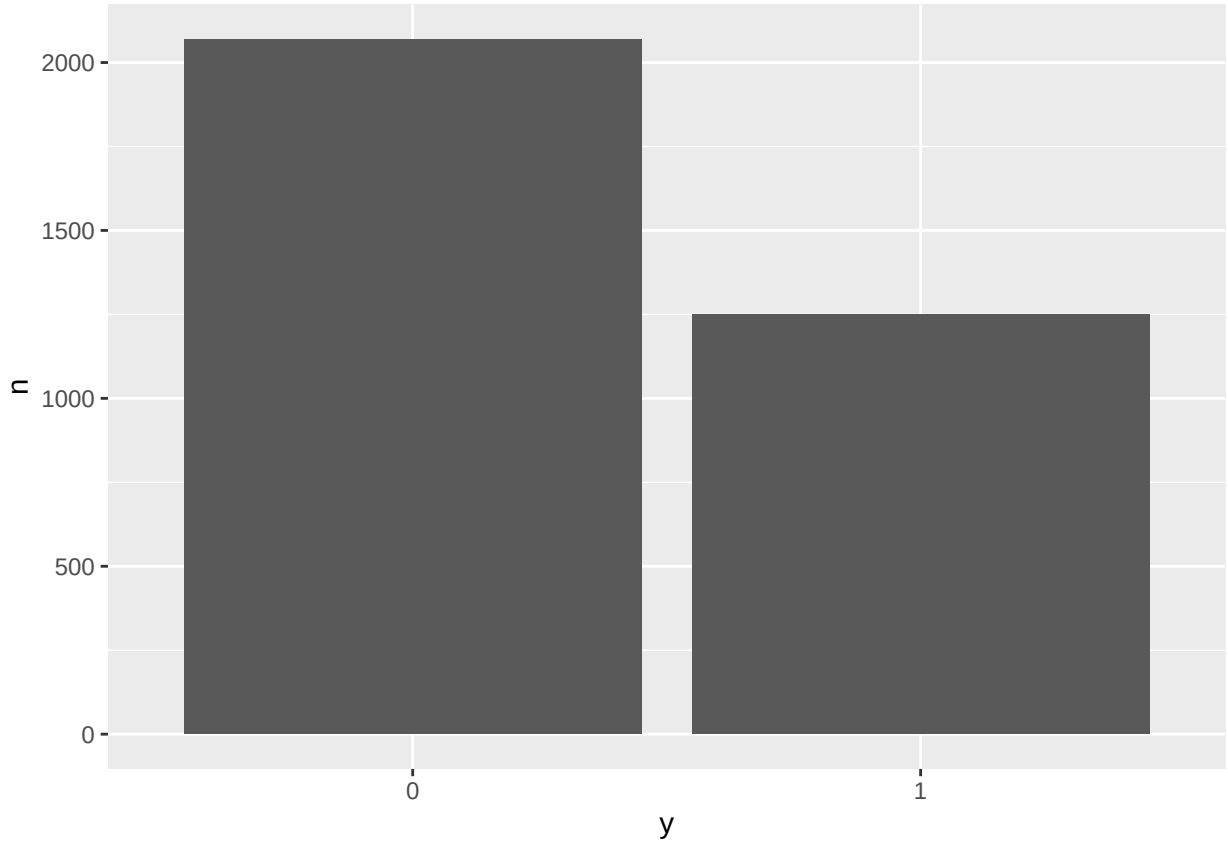
First we look for missing values.

```
library(visdat)
vis_dat(data_bing)
```



We remove NA's and look at the distribution between the two classes.

```
data_bing %<>%
  drop_na() %>%
  rename(y = label) %>%
  select(-name)
data_bing$y <- as.factor(data_bing$y)
data_bing %>%
  count(y) %>%
  ggplot(aes(x = y, y = n)) +
  geom_col()
```



We can see that negative sentiment is much more represented than positive sentiment, so we have to do some down or upsampling.

We will create three different receipes: one using embedding, one using tf-idf and one using Hash.

So we load the embeddings using the "textdata" package.

```
library(textdata)
glove6b <- embedding_glove6b(dimensions = 100)
```

We create a training and test dataset using strata=y to get the same ratio between the classes in both the training and test dataset.

```
library(rsample)
set.seed(19)
tidy_split <- initial_split(data_bing, strata = y)
train_data <- training(tidy_split)
test_data <- testing(tidy_split)
```

We use downsampling only on the training data to better fit the model

```
train_data <- recipe(y~., data = train_data) %>%
  themis::step_downsample(y) %>%
  prep() %>%
  juice()
```

```
## Registered S3 methods overwritten by 'themis':
```

```
##   method                  from
##   bake.step_downsample    recipes
##   bake.step_upsample      recipes
##   prep.step_downsample    recipes
##   prep.step_upsample      recipes
##   tidy.step_downsample    recipes
##   tidy.step_upsample      recipes
##   tunable.step_downsample recipes
##   tunable.step_upsample   recipes
```

```
train_data %>%
  count(y)
```

```
## # A tibble: 2 x 2
##   y         n
##   <fct> <int>
## 1 0       936
## 2 1       936
```

And can now see that the classes are evenly distributed.

We create the three recipies we want to use.

```
library(textrecipes)
tf_idf_rec <- recipe(y~., data = train_data) %>%
  step_tokenize(text) %>%
  step_stem(text) %>%
  step_stopwords(text) %>%
  step_tokenfilter(text, max_tokens = 1000) %>%
  step_tfidf(all_predictors())
embeddings_rec <- recipe(y~., data = train_data) %>%
  step_tokenize(text) %>%
  step_stem(text) %>%
  step_stopwords(text) %>%
  step_tokenfilter(text, max_tokens = 1000) %>%
  step_word_embeddings(text, embeddings = embedding_glove6b())
hash_rec <- recipe(y~., data = train_data) %>%
  step_tokenize(text) %>%
  step_stem(text) %>%
  step_stopwords(text) %>%
  step_tokenfilter(text, max_tokens = 1000) %>%
  step_texthash(text, num_terms = 100)
```

Define models Term frequency

We define three models:

We set some of the parameters for tuning.

Logistic model

```
#model_lg <- logistic_reg(mode = 'classification', penalty = tune(), mixture = 0.5) %>%
  #set_engine('glm', family = binomial)
model_lg <- logistic_reg(mode = 'classification') %>%
  set_engine('glm', family = binomial)
```

KNN model

```
model_knn <- nearest_neighbor(neighbors = tune()) %>%
  set_engine("kknn") %>%
  set_mode("classification")
```

Random Forrest

```
model_rf <-
  rand_forest(trees = NULL, mtry = NULL, min_n = NULL) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")
```

Decision tree

```
model_dt <- decision_tree(mode = 'classification',
                          cost_complexity = tune(),
                          tree_depth = tune(),
                          min_n = tune()
                          ) %>%
  set_engine('rpart')
```

Workflow

We create workflows for each recipe.

tf_idf

```
workflow_general_tf <- workflow() %>%
  add_recipe(tf_idf_rec)
workflow_lg_tf <- workflow_general_tf %>%
  add_model(model_lg)
workflow_knn_tf <- workflow_general_tf %>%
  add_model(model_knn)
workflow_rf_tf <- workflow_general_tf %>%
  add_model(model_rf)
workflow_dt_tf <- workflow_general_tf %>%
  add_model(model_dt)
```

Embeding

```
workflow_general_emb <- workflow() %>%
  add_recipe(embeddings_rec)
workflow_lg_emb <- workflow_general_emb %>%
  add_model(model_lg)
workflow_knn_emb <- workflow_general_emb %>%
  add_model(model_knn)
workflow_rf_emb <- workflow_general_emb %>%
  add_model(model_rf)
workflow_dt_emb <- workflow_general_emb %>%
  add_model(model_dt)
```

hash

```
workflow_general_hash <- workflow() %>%
  add_recipe(hash_rec)
workflow_lg_hash <- workflow_general_hash %>%
  add_model(model_lg)
workflow_knn_hash <- workflow_general_hash %>%
  add_model(model_knn)
workflow_rf_hash <- workflow_general_hash %>%
  add_model(model_rf)
workflow_dt_hash <- workflow_general_hash %>%
  add_model(model_dt)
```

Hyper tuneing

We use vfold_cv to create resampled data. to perfrom hypertuning and fitting.

```
set.seed(100)
k_folds_data <- train_data %>%
  vfold_cv(strata = y,
           v = 3,
           repeats = 3)
```

Define Grids

We define the grids we want to use for the hypertuning

```
logistic_grid <- 5
knn_grid <- 5
dt_grid <- 5
rf_grid <- 5
```

The level defines the amount of parameters that should be considered.

Define tuning process

We define which measures we want to be able to choose best parameters from.

```
model_control <- control_grid(save_pred = TRUE)
model_metrics <- metric_set(accuracy, sens, spec, mn_log_loss, roc_auc)
```

Tune Models

We tune the three different models

```
library(text2vec)
```

```
##
## Vedhæfter pakke: 'text2vec'

## De følgende objekter er maskerede fra 'package:keras':
##
##     fit, normalize

## Det følgende objekt er maskeret fra 'package:infer':
##
##     fit

## Det følgende objekt er maskeret fra 'package:parsnip':
##
##     fit
```

```
# Tune hash models
knn_hash_res <- tune_grid(
  model_knn,
  hash_rec,
  grid = knn_grid,
  control = model_control,
  metrics = model_metrics,
  resamples = k_folds_data
)
dt_hash_res <- tune_grid(
  model_dt,
  hash_rec,
  grid = dt_grid,
  control = model_control,
  metrics = model_metrics,
  resamples = k_folds_data
)
```

```
# Tune embed models
knn_embed_res <- tune_grid(
  model_knn,
  embeddings_rec,
  grid = knn_grid,
```

```
  control = model_control,
  metrics = model_metrics,
  resamples = k_folds_data
)
dt_embed_res <- tune_grid(
  model_dt,
  embeddings_rec,
  grid = dt_grid,
  control = model_control,
  metrics = model_metrics,
  resamples = k_folds_data
)
```

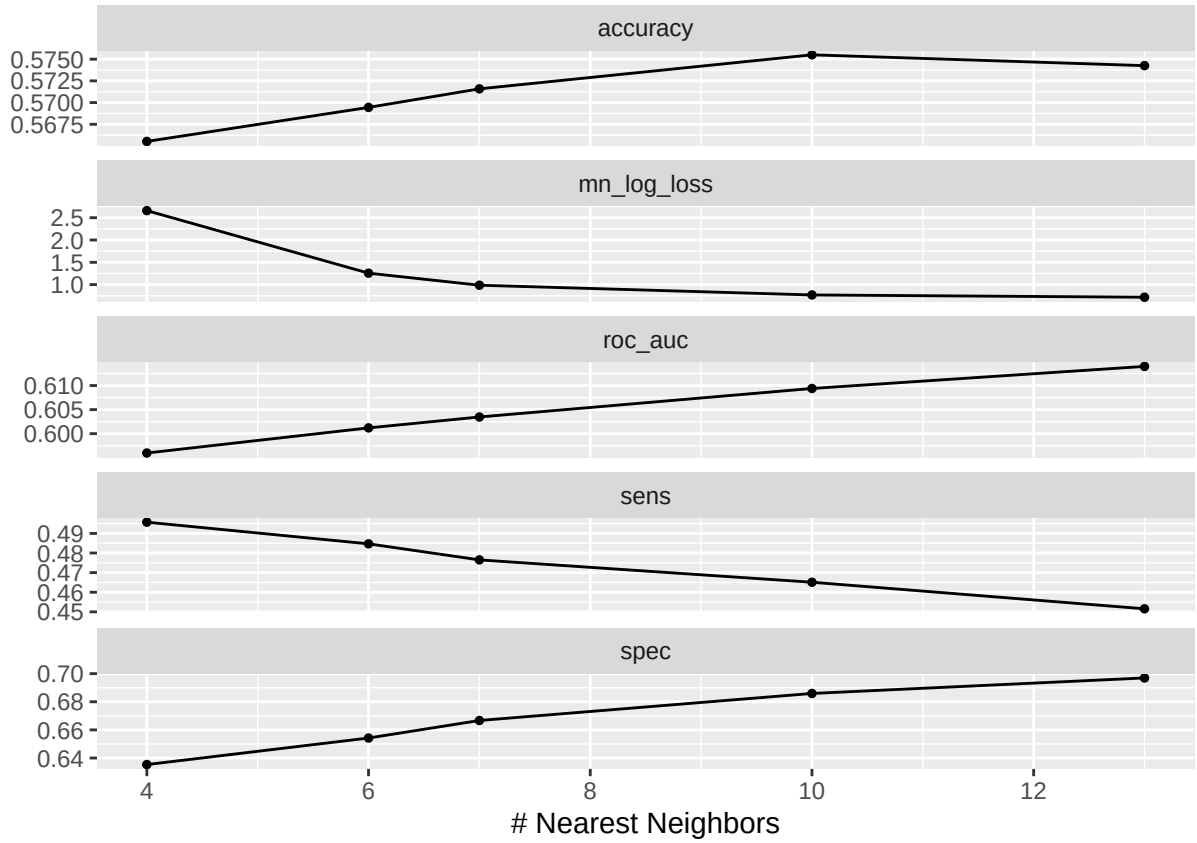
```
# Tune tf-idf models
knn_tf_res <- tune_grid(
  model_knn,
  tf_idf_rec,
  grid = knn_grid,
  control = model_control,
  metrics = model_metrics,
  resamples = k_folds_data
)
dt_tf_res <- tune_grid(
  model_dt,
  tf_idf_rec,
  grid = dt_grid,
  control = model_control,
  metrics = model_metrics,
  resamples = k_folds_data
)
```

Best parameters

We look at the different optimizations and choose the best parameters.

knn_embed_res

```
knn_hash_res %>%
    autoplot()
```

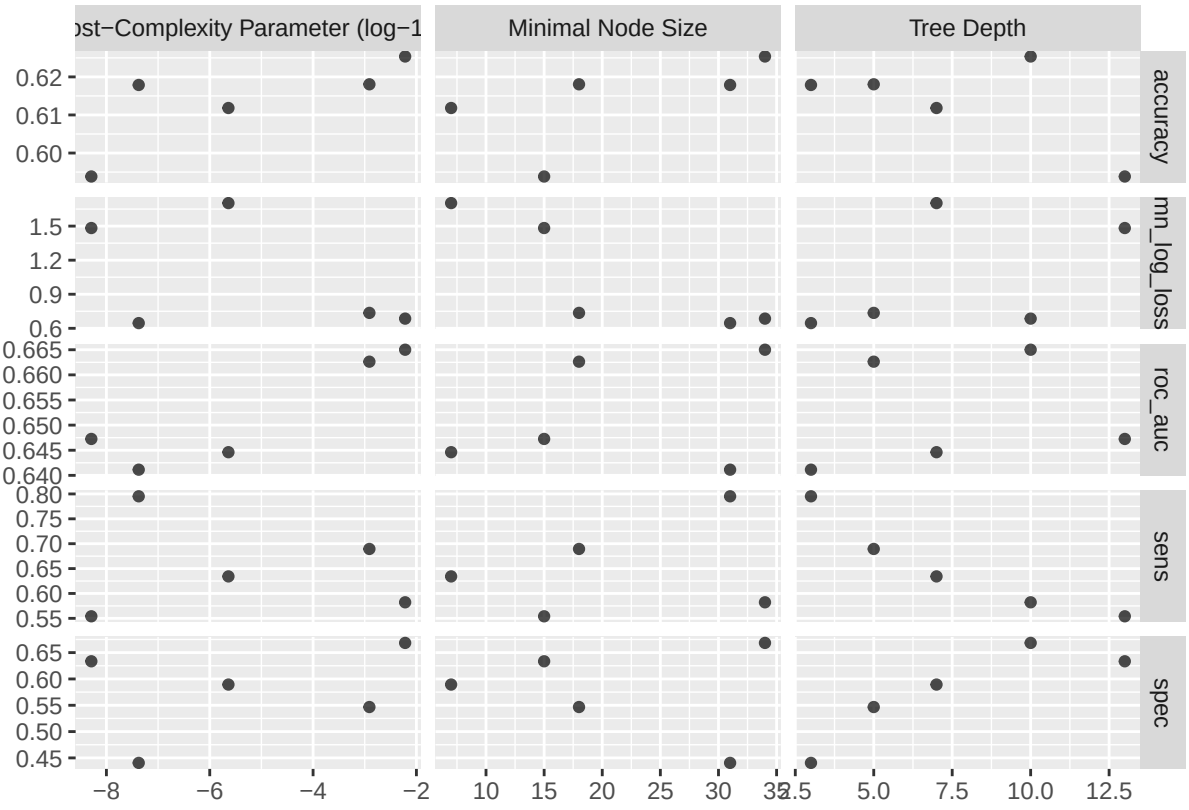



```
best_param_knn_hash_res <- knn_hash_res %>% select_best(metric = 'accuracy')
best_param_knn_hash_res
```

```
## # A tibble: 1 x 2
##   neighbors .config
##       <int> <chr>
## 1        10 Preprocessor1_Model4
```

decision tree hash

```
dt_hash_res %>%
    autoplot()
```

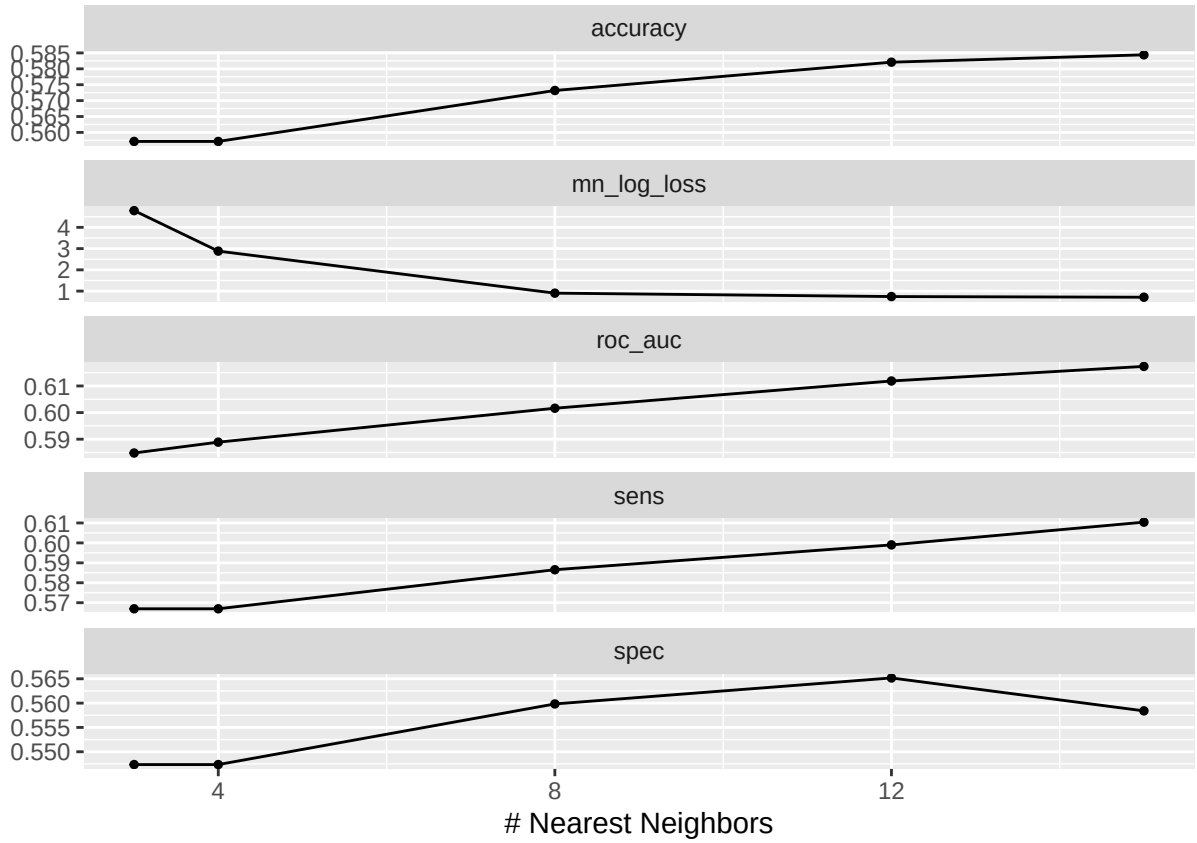



```
best_param_dt_hash_res <- dt_hash_res %>% select_best(metric = 'accuracy')
best_param_dt_hash_res
```

```
## # A tibble: 1 x 4
##   cost_complexity tree_depth min_n .config
##             <dbl>      <int> <int> <chr>
## 1         0.00606         10    34 Preprocessor1_Model5
```

knn_embed_res

```
knn_embed_res %>%
    autoplot()
```

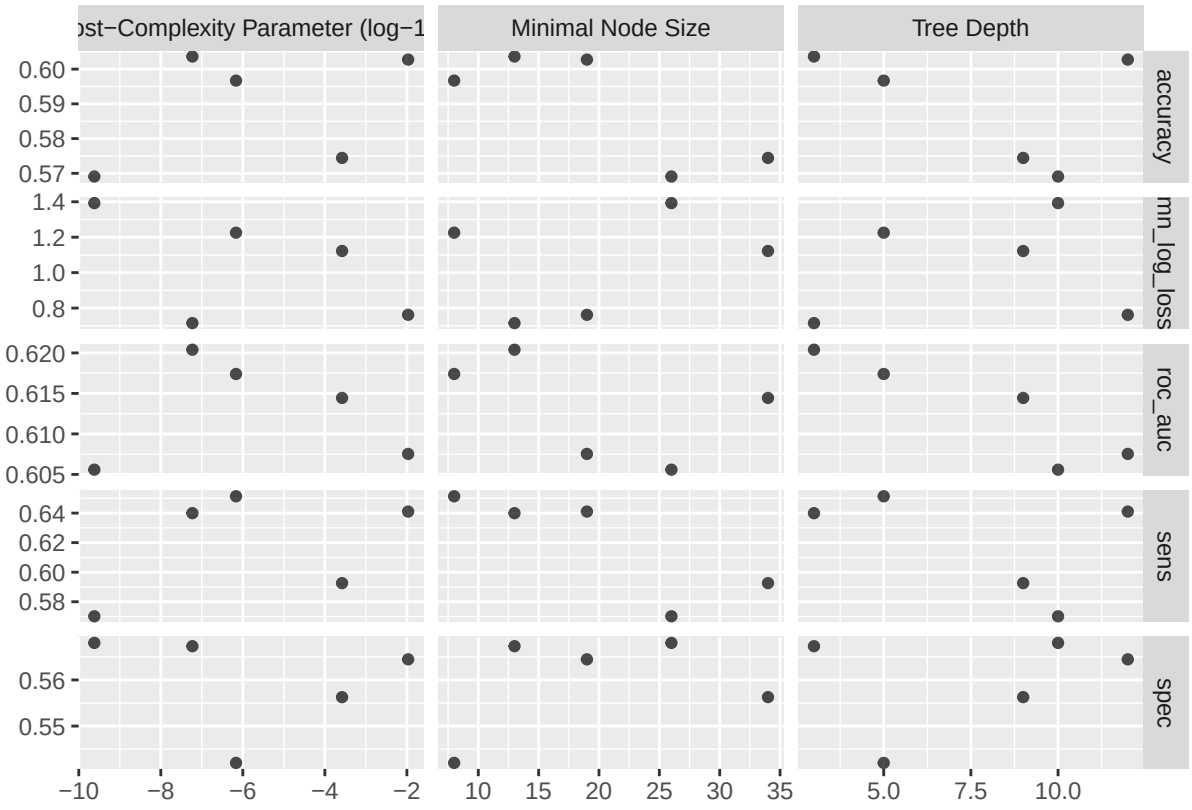



```
best_param_knn_embed_res <- knn_embed_res %>% select_best(metric = 'accuracy')
best_param_knn_embed_res
```

```
## # A tibble: 1 x 2
##   neighbors .config
##       <int> <chr>
## 1        15 Preprocessor1_Model5
```

dt_embed_res

```
dt_embed_res %>%
    autoplot()
```

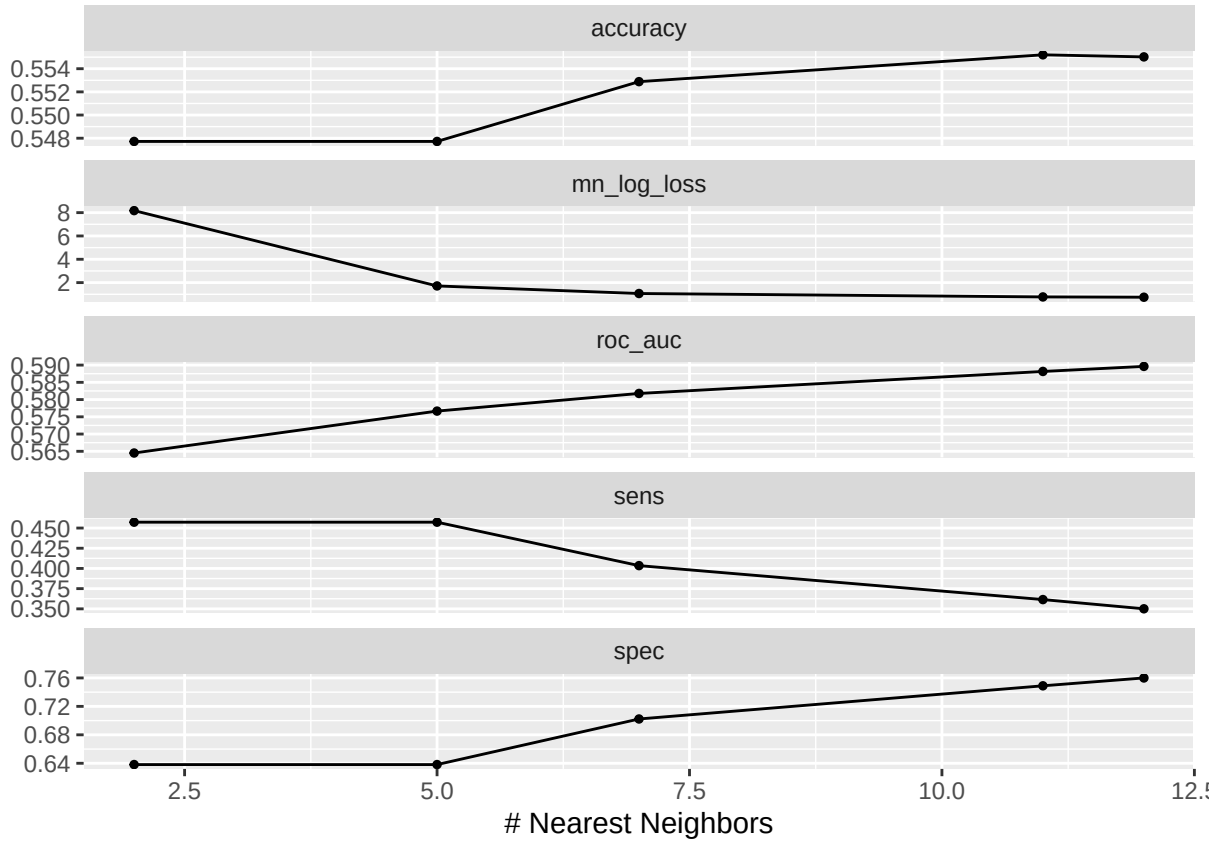



```
best_param_dt_embed_res <- dt_embed_res %>% select_best(metric = 'accuracy')
best_param_dt_embed_res
```

```
## # A tibble: 1 x 4
##   cost_complexity tree_depth min_n .config
##             <dbl>      <int> <int> <chr>
## 1    0.0000000589          3    13 Preprocessor1_Model4
```

knn_tf_res

```
knn_tf_res %>%
    autoplot()
```

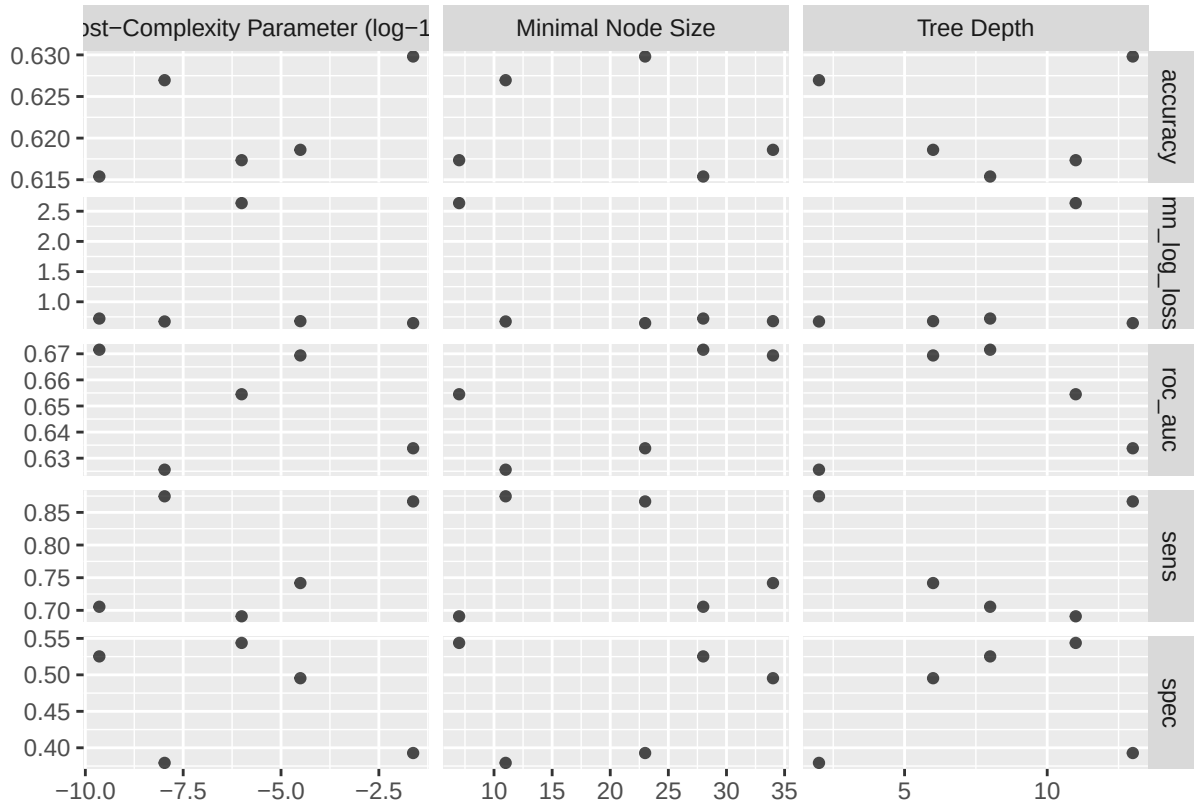



```
best_param_knn_tf_res <- knn_tf_res %>% select_best(metric = 'accuracy')
best_param_knn_tf_res
```

```
## # A tibble: 1 x 2
##   neighbors .config
##       <int> <chr>
## 1        11 Preprocessor1_Model4
```

dt_tf_res

```
dt_tf_res %>%
    autoplot()
```




```
best_param_dt_tf_res <- dt_tf_res %>% select_best(metric = 'accuracy')
best_param_dt_tf_res
```

```
## # A tibble: 1 x 4
##   cost_complexity tree_depth min_n .config
##             <dbl>      <int> <int> <chr>
## 1          0.0238         13    23 Preprocessor1_Model1
```

Finalize workflows

We now fit the best parameters into the workflow of the two models that needed hypertuning.

Hash

```
workflow_final_knn_hash <- workflow_knn_hash %>%
  finalize_workflow(parameters = best_param_knn_hash_res)
#workflow_final_rf_hash <- workflow_rf_hash %>%
 # finalize_workflow(parameters = best_param_rf_hash_res)
workflow_final_dt_hash <- workflow_dt_hash %>%
  finalize_workflow(parameters = best_param_dt_hash_res)
```

Tf-idf

```
workflow_final_knn_tf <- workflow_knn_tf %>%
  finalize_workflow(parameters = best_param_knn_tf_res)
#workflow_final_rf_tf <- workflow_rf_tf %>%
 # finalize_workflow(parameters = best_param_rf_tf_res)
workflow_final_dt_tf <- workflow_dt_tf %>%
  finalize_workflow(parameters = best_param_dt_tf_res)
```

Embedings

```
workflow_final_knn_emb <- workflow_knn_emb %>%
  finalize_workflow(parameters = best_param_knn_embed_res)
#workflow_final_rf_emb <- workflow_rf_emb %>%
 # finalize_workflow(parameters = best_param_rf_embed_res)
workflow_final_dt_emb <- workflow_dt_emb %>%
  finalize_workflow(parameters = best_param_dt_embed_res)
```

Evaluate models

here we us the resampled data to evaluate the models.

Logistic regression

```


log_res_hash <-
  workflow_lg_hash %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
```

hash

```
## ! Fold1, Repeat1: preprocessor 1/1, model 1/1: glm.fit: fitted probabilities numerically 0...

## ! Fold2, Repeat1: preprocessor 1/1, model 1/1: glm.fit: fitted probabilities numerically 0...

## ! Fold1, Repeat2: preprocessor 1/1, model 1/1: glm.fit: fitted probabilities numerically 0...

## ! Fold2, Repeat2: preprocessor 1/1, model 1/1: glm.fit: fitted probabilities numerically 0...
```

```
## ! Fold2, Repeat3: preprocessor 1/1, model 1/1: glm.fit: fitted probabilities numerically 0...

## ! Fold3, Repeat3: preprocessor 1/1, model 1/1: glm.fit: fitted probabilities numerically 0...
```

```
log_res_hash %>% collect_metrics(summarize = TRUE)
```

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.661     9 0.0103  Preprocessor1_Model1
## 2 f_meas    binary     0.670     9 0.0106  Preprocessor1_Model1
## 3 kap       binary     0.322     9 0.0205  Preprocessor1_Model1
## 4 precision binary     0.652     9 0.00919 Preprocessor1_Model1
## 5 recall    binary     0.689     9 0.0131  Preprocessor1_Model1
## 6 roc_auc   binary     0.719     9 0.0107  Preprocessor1_Model1
## 7 sens      binary     0.689     9 0.0131  Preprocessor1_Model1
## 8 spec      binary     0.632     9 0.00988 Preprocessor1_Model1
```

```
log_res_tf <-
  workflow_lg_tf %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
```

Tf_idf

```
## ! Fold1, Repeat1: preprocessor 1/1, model 1/1: glm.fit: algorithm did not converge, glm.fi...

## ! Fold1, Repeat1: preprocessor 1/1, model 1/1 (predictions): prediction from a rank-defici...

## ! Fold2, Repeat1: preprocessor 1/1, model 1/1: glm.fit: algorithm did not converge, glm.fi...

## ! Fold2, Repeat1: preprocessor 1/1, model 1/1 (predictions): prediction from a rank-defici...

## ! Fold3, Repeat1: preprocessor 1/1, model 1/1: glm.fit: algorithm did not converge, glm.fi...

## ! Fold3, Repeat1: preprocessor 1/1, model 1/1 (predictions): prediction from a rank-defici...

## ! Fold1, Repeat2: preprocessor 1/1, model 1/1: glm.fit: algorithm did not converge, glm.fi...

## ! Fold1, Repeat2: preprocessor 1/1, model 1/1 (predictions): prediction from a rank-defici...
```

```
## ! Fold2, Repeat2: preprocessor 1/1, model 1/1: glm.fit: algorithm did not converge, glm.fi...

## ! Fold2, Repeat2: preprocessor 1/1, model 1/1 (predictions): prediction from a rank-defici...

## ! Fold3, Repeat2: preprocessor 1/1, model 1/1: glm.fit: algorithm did not converge, glm.fi...

## ! Fold3, Repeat2: preprocessor 1/1, model 1/1 (predictions): prediction from a rank-defici...

## ! Fold1, Repeat3: preprocessor 1/1, model 1/1: glm.fit: algorithm did not converge, glm.fi...

## ! Fold1, Repeat3: preprocessor 1/1, model 1/1 (predictions): prediction from a rank-defici...

## ! Fold2, Repeat3: preprocessor 1/1, model 1/1: glm.fit: algorithm did not converge, glm.fi...

## ! Fold2, Repeat3: preprocessor 1/1, model 1/1 (predictions): prediction from a rank-defici...

## ! Fold3, Repeat3: preprocessor 1/1, model 1/1: glm.fit: algorithm did not converge, glm.fi...

## ! Fold3, Repeat3: preprocessor 1/1, model 1/1 (predictions): prediction from a rank-defici...
```

```
log_res_tf %>% collect_metrics(summarize = TRUE)
```

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.616     9 0.00435 Preprocessor1_Model1
## 2 f_meas    binary     0.617     9 0.00405 Preprocessor1_Model1
## 3 kap       binary     0.233     9 0.00870 Preprocessor1_Model1
## 4 precision binary     0.617     9 0.00496 Preprocessor1_Model1
## 5 recall    binary     0.617     9 0.00585 Preprocessor1_Model1
## 6 roc_auc   binary     0.640     9 0.00591 Preprocessor1_Model1
## 7 sens      binary     0.617     9 0.00585 Preprocessor1_Model1
## 8 spec      binary     0.616     9 0.00849 Preprocessor1_Model1
```

```
log_res_emb <-
  workflow_lg_emb %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
log_res_emb %>% collect_metrics(summarize = TRUE)
```

Embeding

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.665     9 0.00561 Preprocessor1_Model1
## 2 f_meas    binary     0.665     9 0.00631 Preprocessor1_Model1
## 3 kap       binary     0.329     9 0.0112  Preprocessor1_Model1
## 4 precision binary     0.664     9 0.00599 Preprocessor1_Model1
## 5 recall    binary     0.667     9 0.00942 Preprocessor1_Model1
## 6 roc_auc   binary     0.727     9 0.00559 Preprocessor1_Model1
## 7 sens      binary     0.667     9 0.00942 Preprocessor1_Model1
## 8 spec      binary     0.662     9 0.00875 Preprocessor1_Model1
```

KNN model

```
knn_res_hash <-
  workflow_final_knn_hash %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
knn_res_hash %>% collect_metrics(summarize = TRUE)
```

Hash

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.575     9 0.00609 Preprocessor1_Model1
## 2 f_meas    binary     0.523     9 0.00739 Preprocessor1_Model1
## 3 kap       binary     0.151     9 0.0122  Preprocessor1_Model1
## 4 precision binary     0.597     9 0.00800 Preprocessor1_Model1
## 5 recall    binary     0.465     9 0.00835 Preprocessor1_Model1
## 6 roc_auc   binary     0.609     9 0.00501 Preprocessor1_Model1
## 7 sens      binary     0.465     9 0.00835 Preprocessor1_Model1
## 8 spec      binary     0.686     9 0.00865 Preprocessor1_Model1
```

```
knn_res_tf <-
  workflow_final_knn_tf %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
```

```
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
knn_res_tf %>% collect_metrics(summarize = TRUE)
```

TF-idf

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.555     9 0.00589 Preprocessor1_Model1
## 2 f_meas    binary     0.441     9 0.0240  Preprocessor1_Model1
## 3 kap       binary     0.110     9 0.0118  Preprocessor1_Model1
## 4 precision binary     0.595     9 0.0115  Preprocessor1_Model1
## 5 recall    binary     0.361     9 0.0326  Preprocessor1_Model1
## 6 roc_auc   binary     0.588     9 0.00680 Preprocessor1_Model1
## 7 sens      binary     0.361     9 0.0326  Preprocessor1_Model1
## 8 spec      binary     0.749     9 0.0299  Preprocessor1_Model1
```

```
knn_res_emb <-
  workflow_final_knn_emb %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
knn_res_emb %>% collect_metrics(summarize = TRUE)
```

Embedings

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.584     9 0.00882 Preprocessor1_Model1
## 2 f_meas    binary     0.595     9 0.00591 Preprocessor1_Model1
## 3 kap       binary     0.169     9 0.0176  Preprocessor1_Model1
## 4 precision binary     0.581     9 0.00931 Preprocessor1_Model1
## 5 recall    binary     0.610     9 0.00478 Preprocessor1_Model1
## 6 roc_auc   binary     0.617     9 0.00704 Preprocessor1_Model1
## 7 sens      binary     0.610     9 0.00478 Preprocessor1_Model1
## 8 spec      binary     0.558     9 0.0170  Preprocessor1_Model1
```

Random forest model

```
rf_res_hash <-
  workflow_rf_hash %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
rf_res_hash %>% collect_metrics(summarize = TRUE)
```

hash

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.687     9 0.00660 Preprocessor1_Model1
## 2 f_meas    binary     0.706     9 0.00742 Preprocessor1_Model1
## 3 kap       binary     0.374     9 0.0132  Preprocessor1_Model1
## 4 precision binary     0.666     9 0.00513 Preprocessor1_Model1
## 5 recall    binary     0.751     9 0.0124  Preprocessor1_Model1
## 6 roc_auc   binary     0.758     9 0.00913 Preprocessor1_Model1
## 7 sens      binary     0.751     9 0.0124  Preprocessor1_Model1
## 8 spec      binary     0.623     9 0.00733 Preprocessor1_Model1
```

```
rf_res_tf <-
  workflow_rf_tf %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
rf_res_tf %>% collect_metrics(summarize = TRUE)
```

TF-idf

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.733     9 0.00700 Preprocessor1_Model1
## 2 f_meas    binary     0.744     9 0.00740 Preprocessor1_Model1
## 3 kap       binary     0.467     9 0.0140  Preprocessor1_Model1
```

```
## 4 precision binary     0.715     9 0.00595 Preprocessor1_Model1
## 5 recall    binary     0.775     9 0.0105  Preprocessor1_Model1
## 6 roc_auc   binary     0.818     9 0.00599 Preprocessor1_Model1
## 7 sens      binary     0.775     9 0.0105  Preprocessor1_Model1
## 8 spec      binary     0.691     9 0.00692 Preprocessor1_Model1
```

```
rf_res_emb <-
  workflow_rf_emb %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
rf_res_emb %>% collect_metrics(summarize = TRUE)
```

Embedings

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.653     9 0.00658 Preprocessor1_Model1
## 2 f_meas    binary     0.657     9 0.00554 Preprocessor1_Model1
## 3 kap       binary     0.306     9 0.0132  Preprocessor1_Model1
## 4 precision binary     0.649     9 0.00796 Preprocessor1_Model1
## 5 recall    binary     0.666     9 0.00597 Preprocessor1_Model1
## 6 roc_auc   binary     0.705     9 0.00485 Preprocessor1_Model1
## 7 sens      binary     0.666     9 0.00597 Preprocessor1_Model1
## 8 spec      binary     0.640     9 0.0117  Preprocessor1_Model1
```

Decision tree

```
dt_res_hash <-
  workflow_final_dt_hash %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
dt_res_hash %>% collect_metrics(summarize = TRUE)
```

hash

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.625     9 0.00965 Preprocessor1_Model1
## 2 f_meas    binary     0.607     9 0.0152  Preprocessor1_Model1
## 3 kap       binary     0.251     9 0.0193  Preprocessor1_Model1
## 4 precision binary     0.636     9 0.00802 Preprocessor1_Model1
## 5 recall    binary     0.582     9 0.0227  Preprocessor1_Model1
## 6 roc_auc   binary     0.665     9 0.00712 Preprocessor1_Model1
## 7 sens      binary     0.582     9 0.0227  Preprocessor1_Model1
## 8 spec      binary     0.668     9 0.0106  Preprocessor1_Model1
```

```
dt_res_tf <-
  workflow_final_dt_tf %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
dt_res_tf %>% collect_metrics(summarize = TRUE)
```

Tf_idf

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.630     9 0.00400 Preprocessor1_Model1
## 2 f_meas    binary     0.700     9 0.00431 Preprocessor1_Model1
## 3 kap       binary     0.260     9 0.00799 Preprocessor1_Model1
## 4 precision binary     0.589     9 0.00416 Preprocessor1_Model1
## 5 recall    binary     0.867     9 0.0166  Preprocessor1_Model1
## 6 roc_auc   binary     0.634     9 0.00530 Preprocessor1_Model1
## 7 sens      binary     0.867     9 0.0166  Preprocessor1_Model1
## 8 spec      binary     0.393     9 0.0191  Preprocessor1_Model1
```

```
dt_res_emb <-
  workflow_final_dt_emb %>%
  fit_resamples(
    resamples = k_folds_data,
    metrics = metric_set(
      recall, precision, f_meas,
      accuracy, kap,
```

```
      roc_auc, sens, spec),
    control = control_resamples(
      save_pred = TRUE)
    )
dt_res_emb %>% collect_metrics(summarize = TRUE)
```

Embeding

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.604     9 0.00355 Preprocessor1_Model1
## 2 f_meas    binary     0.612     9 0.0181  Preprocessor1_Model1
## 3 kap       binary     0.207     9 0.00711 Preprocessor1_Model1
## 4 precision binary     0.600     9 0.00659 Preprocessor1_Model1
## 5 recall    binary     0.640     9 0.0407  Preprocessor1_Model1
## 6 roc_auc   binary     0.620     9 0.00714 Preprocessor1_Model1
## 7 sens      binary     0.640     9 0.0407  Preprocessor1_Model1
## 8 spec      binary     0.567     9 0.0357  Preprocessor1_Model1
```

Compare performance

We get a summary for the performed models. We add the model name to each metric to keep the models appart from each other later on.

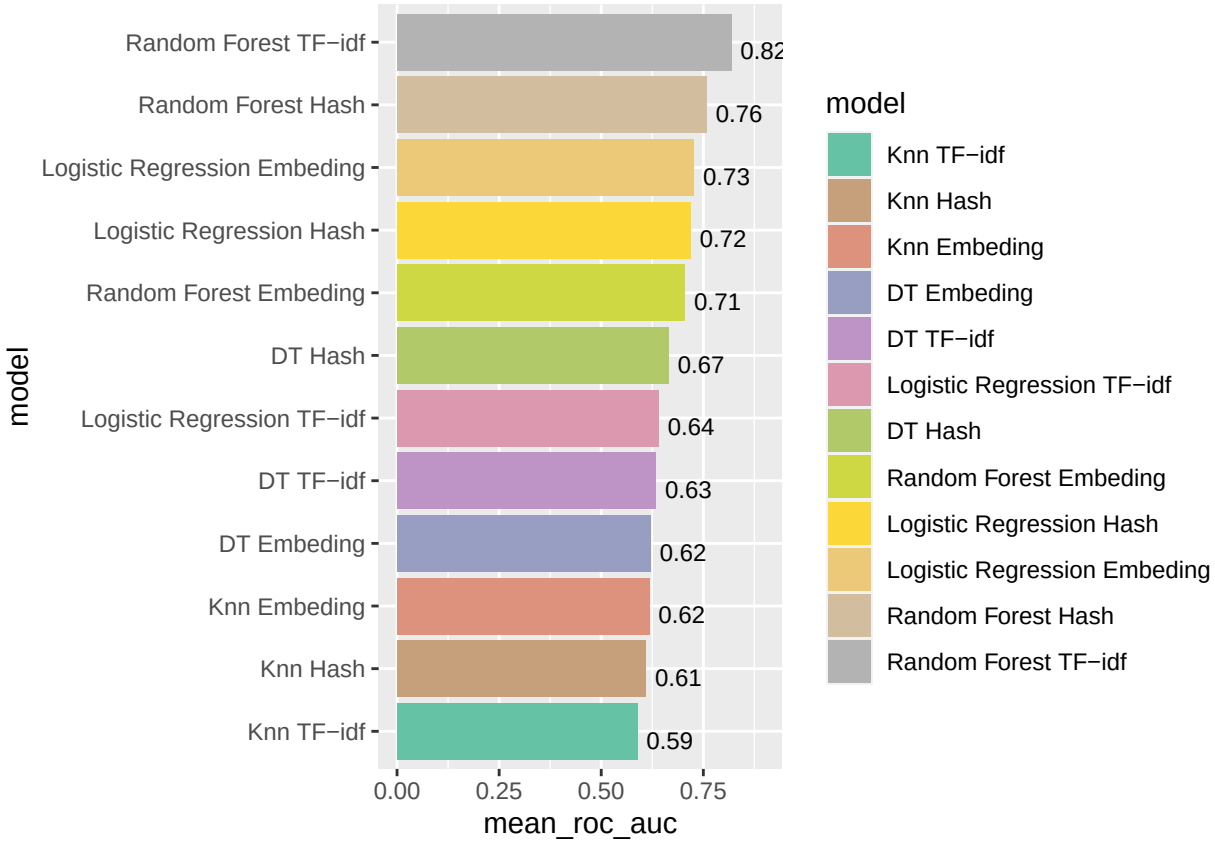
```
log_metrics_tf <-
  log_res_tf %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "Logistic Regression TF-idf")
log_metrics_emb <-
  log_res_emb %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "Logistic Regression Embeding")
log_metrics_hash <-
  log_res_hash %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "Logistic Regression Hash")
rf_metrics_tf <-
  rf_res_tf %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "Random Forest TF-idf")
rf_metrics_emb <-
  rf_res_emb %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "Random Forest Embeding")
rf_metrics_hash <-
  rf_res_hash %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "Random Forest Hash")
knn_metrics_tf <-
  knn_res_tf %>%
  collect_metrics(summarise = TRUE) %>%
```

```
  mutate(model = "Knn TF-idf")
knn_metrics_emb <-
  knn_res_emb %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "Knn Embeding")
knn_metrics_hash <-
  knn_res_hash %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "Knn Hash")
dt_metrics_tf <-
  dt_res_tf %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "DT TF-idf")
dt_metrics_emb <-
  dt_res_emb %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "DT Embeding")
dt_metrics_hash <-
  dt_res_hash %>%
  collect_metrics(summarise = TRUE) %>%
  mutate(model = "DT Hash")
```

```
model_compare <- bind_rows(
                          log_metrics_tf,
                          log_metrics_emb,
                          log_metrics_hash,
                          rf_metrics_tf,
                          rf_metrics_emb,
                          rf_metrics_hash,
                          knn_metrics_tf,
                          knn_metrics_emb,
                          knn_metrics_hash,
                          dt_metrics_tf,
                          dt_metrics_emb,
                          dt_metrics_hash
                           )
model_comp <-
  model_compare %>%
  select(model, .metric, mean, std_err) %>%
  pivot_wider(names_from = .metric, values_from = c(mean, std_err))
library(RColorBrewer)
nb.cols <- 12
mycolors <- colorRampPalette(brewer.pal(8, "Set2"))(nb.cols)
model_comp %>%
  arrange(mean_f_meas) %>%
  mutate(model = fct_reorder(model, mean_f_meas)) %>%
  ggplot(aes(model, mean_f_meas, fill=model)) +
  geom_col() +
  coord_flip() +
  scale_fill_manual(values = mycolors) +
  #scale_fill_brewer(palette = "Blues") +
   geom_text(
     size = 3,
```

```
    aes(label = round(mean_f_meas, 2), y = mean_f_meas + 0.08),
    vjust = 1
  )
```


```
model_comp %>%
  arrange(mean_roc_auc) %>%
  mutate(model = fct_reorder(model, mean_roc_auc)) %>%
  ggplot(aes(model, mean_roc_auc, fill=model)) +
  geom_col() +
  coord_flip() +
  scale_fill_manual(values = mycolors) +
  #scale_fill_brewer(palette = "Blues") +
     geom_text(
     size = 3,
     aes(label = round(mean_roc_auc, 2), y = mean_roc_auc + 0.08),
     vjust = 1
  )
```



Choose model

The best model seems to be Random Forest using TF-idf we also look at the second best model which is random forest using hash.

So we only continue with the two best ones.

Random forest model with TF IDF

Performance metrics Show average performance over all folds:

```
rf_res_tf %>%  collect_metrics(summarize = TRUE)
```

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.733     9 0.00700 Preprocessor1_Model1
## 2 f_meas    binary     0.744     9 0.00740 Preprocessor1_Model1
## 3 kap       binary     0.467     9 0.0140  Preprocessor1_Model1
## 4 precision binary     0.715     9 0.00595 Preprocessor1_Model1
## 5 recall    binary     0.775     9 0.0105  Preprocessor1_Model1
## 6 roc_auc   binary     0.818     9 0.00599 Preprocessor1_Model1
## 7 sens      binary     0.775     9 0.0105  Preprocessor1_Model1
## 8 spec      binary     0.691     9 0.00692 Preprocessor1_Model1
```

Collect model predictions To obtain the actual model predictions, we use the function collect_predictions and save the result as rf_pred_tf:

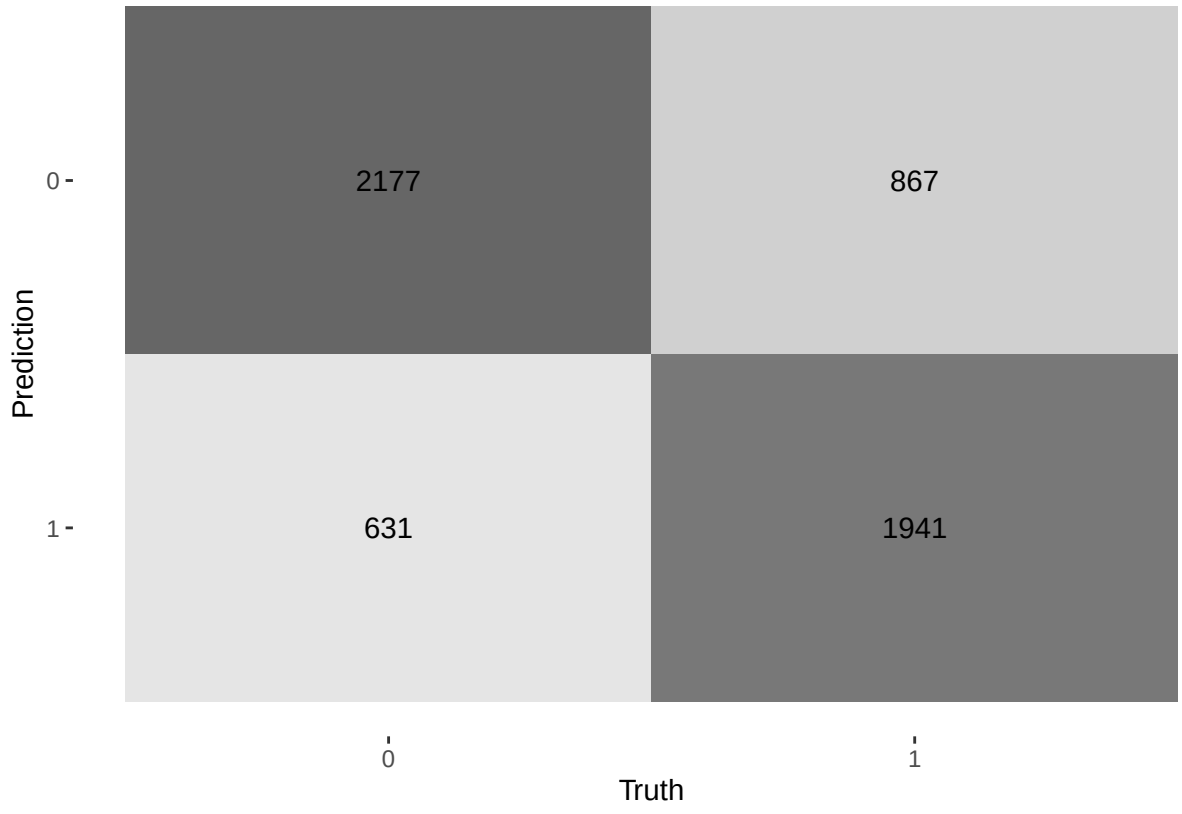
```
rf_pred_tf <-
  rf_res_tf %>%
  collect_predictions()
```

Confusion Matrix We can now use our collected predictions to make a confusion matrix

```
rf_pred_tf %>%
  conf_mat(y, .pred_class)
```

```
##           Truth
## Prediction    0    1
##          0 2177  867
##          1  631 1941
```

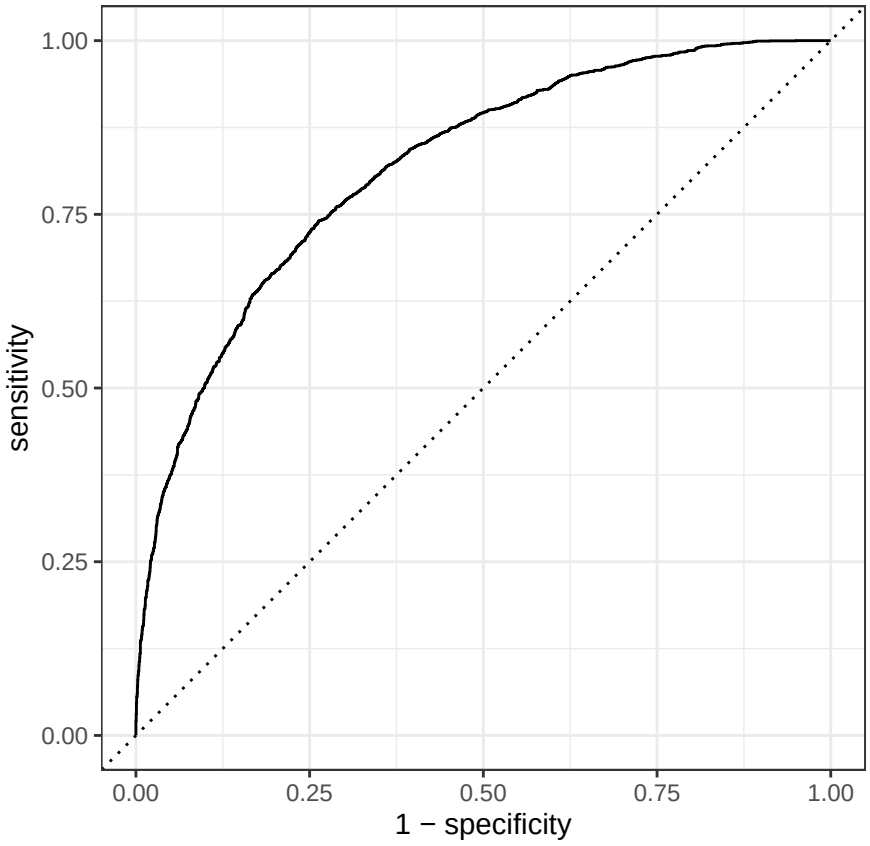
```
rf_pred_tf %>%
  conf_mat(y, .pred_class) %>%
  autoplot(type = "heatmap")
```



We can see the model does okay predicting the correct classes.

ROC curve We will now create the ROC curve with 1 - specificity on the x-axis (false positive fraction = FP/(FP+TN)) and sensitivity on the y axis (true positive fraction = TP/(TP+FN)).

```
rf_pred_tf %>%
  roc_curve(y,.pred_0) %>%
  autoplot()
```



Random forest model hash

Collect model predictions To obtain the actual model predictions, we use the function collect_predictions and save the result as rf_pred_hash:

```
rf_pred_hash <-
  rf_res_hash %>%
  collect_predictions()
```

Performance metrics Show average performance over all folds (note that we use rf_res):

```
rf_res_hash %>%  collect_metrics(summarize = TRUE)
```

```
## # A tibble: 8 x 6
##   .metric   .estimator  mean     n std_err .config
##   <chr>     <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy  binary     0.687     9 0.00660 Preprocessor1_Model1
## 2 f_meas    binary     0.706     9 0.00742 Preprocessor1_Model1
## 3 kap       binary     0.374     9 0.0132  Preprocessor1_Model1
```

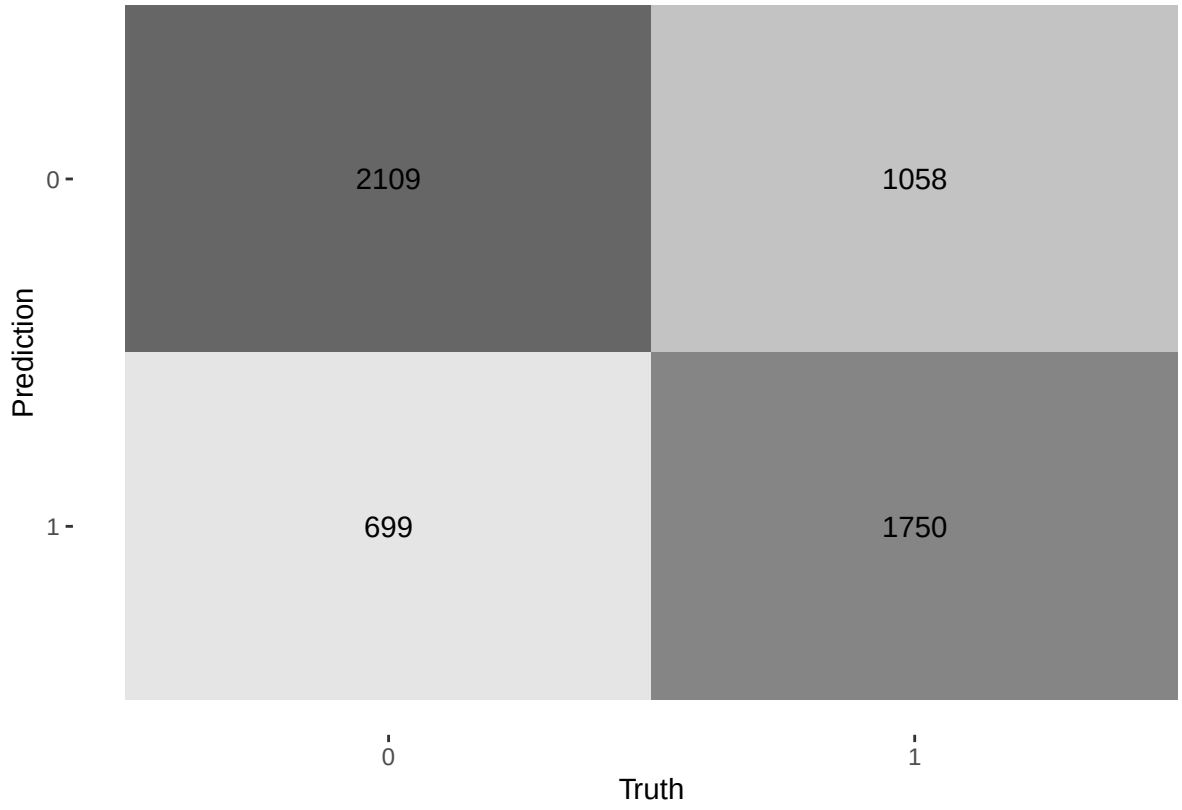
```
## 4 precision binary     0.666     9 0.00513 Preprocessor1_Model1
## 5 recall    binary     0.751     9 0.0124  Preprocessor1_Model1
## 6 roc_auc   binary     0.758     9 0.00913 Preprocessor1_Model1
## 7 sens      binary     0.751     9 0.0124  Preprocessor1_Model1
## 8 spec      binary     0.623     9 0.00733 Preprocessor1_Model1
```

Confusion Matrix We can now use our collected predictions to make a confusion matrix

```
rf_pred_hash %>%
  conf_mat(y, .pred_class)
```

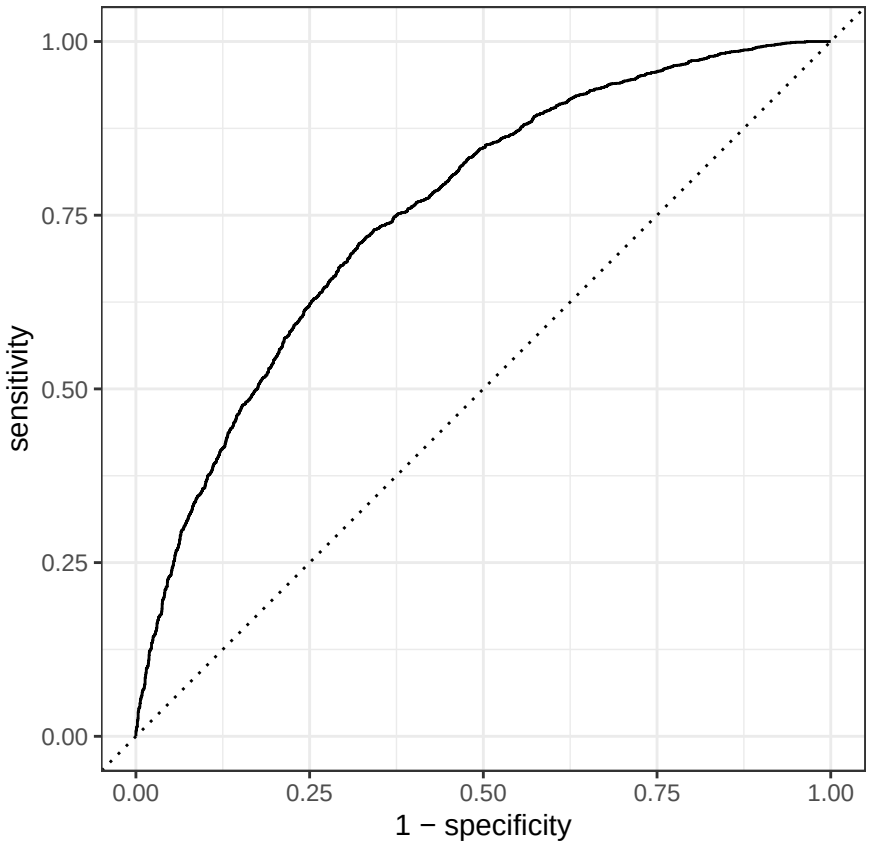
```
##            Truth
## Prediction    0    1
##          0 2109 1058
##          1  699 1750
```

```
rf_pred_hash %>%
  conf_mat(y, .pred_class) %>%
  autoplot(type = "heatmap")
```



ROC curve We will now create the ROC curve with 1 - specificity on the x-axis (false positive fraction = FP/(FP+TN)) and sensitivity on the y axis (true positive fraction = TP/(TP+FN)).

```
rf_pred_hash %>%
  roc_curve(y, .pred_0) %>%
  autoplot()
```



Models on test data

We now want to look at how the two models perform on test data.

Random forest model TF IDF

```
last_fit_rf <- last_fit(workflow_rf_tf,
                        split = tidy_split,
                        metrics = metric_set(
                          recall, precision, f_meas,
                          accuracy, kap,
                          roc_auc, sens, spec)
                        )
```

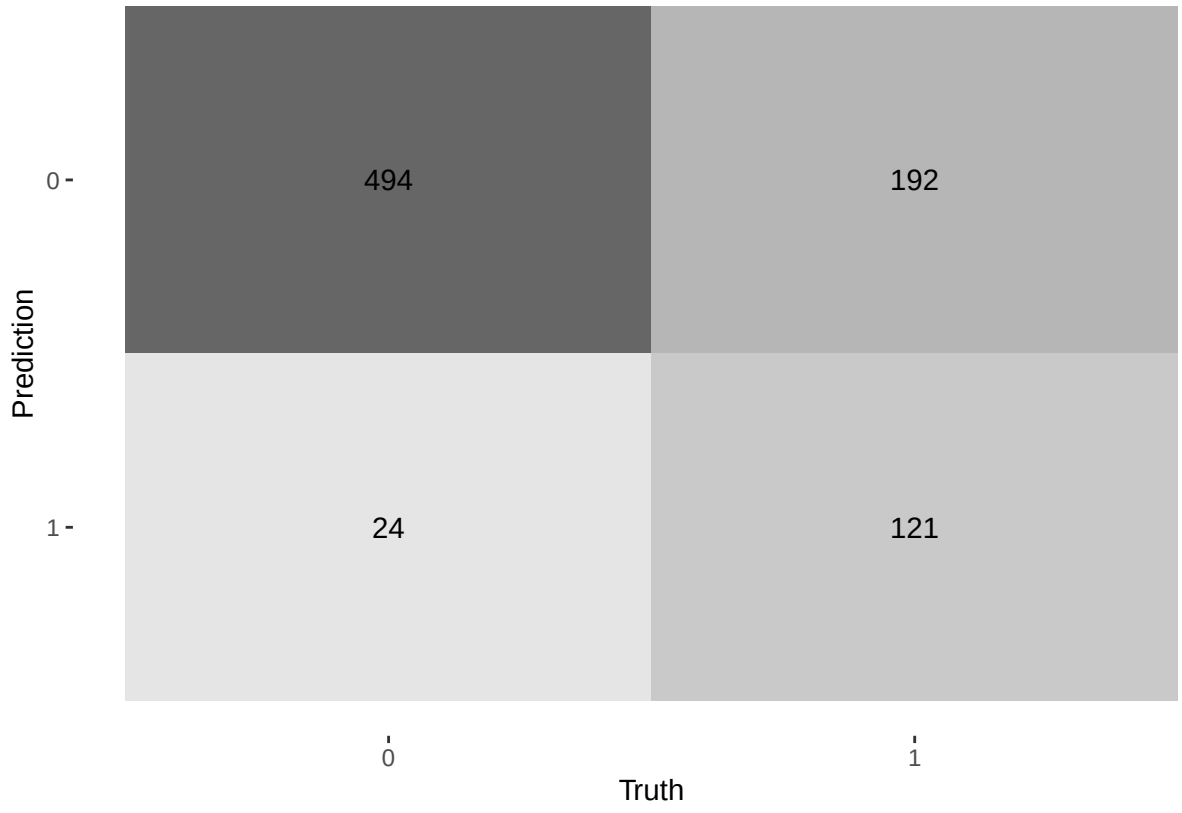
```
last_fit_rf %>%
  collect_metrics()
```

```
## # A tibble: 8 x 4
```

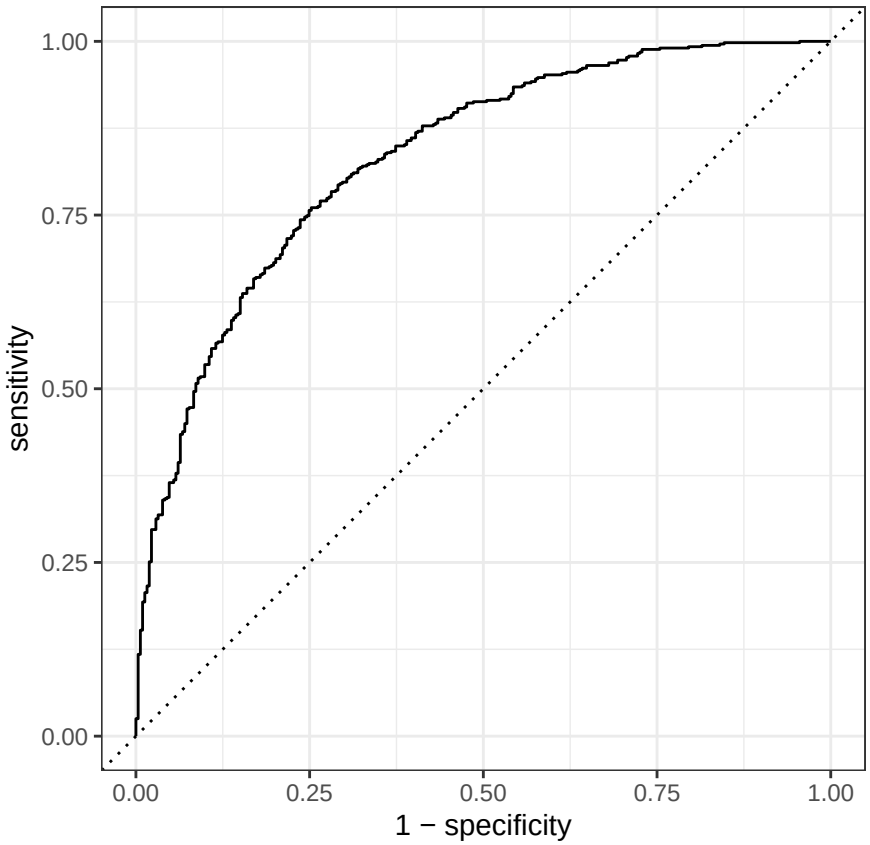
```
##   .metric   .estimator .estimate .config
##   <chr>     <chr>          <dbl> <chr>
## 1 recall    binary         0.954 Preprocessor1_Model1
## 2 precision binary         0.720 Preprocessor1_Model1
## 3 f_meas    binary         0.821 Preprocessor1_Model1
## 4 accuracy  binary         0.740 Preprocessor1_Model1
## 5 kap       binary         0.381 Preprocessor1_Model1
## 6 sens      binary         0.954 Preprocessor1_Model1
## 7 spec      binary         0.387 Preprocessor1_Model1
## 8 roc_auc   binary         0.831 Preprocessor1_Model1
```

We can again make a confusion matrix on the test data predictions

```
last_fit_rf %>%
  collect_predictions() %>%
  conf_mat(y, .pred_class) %>%
  autoplot(type = "heatmap")
```




```
last_fit_rf %>%
  collect_predictions() %>%
  roc_curve(y, .pred_0) %>%
  autoplot()
```



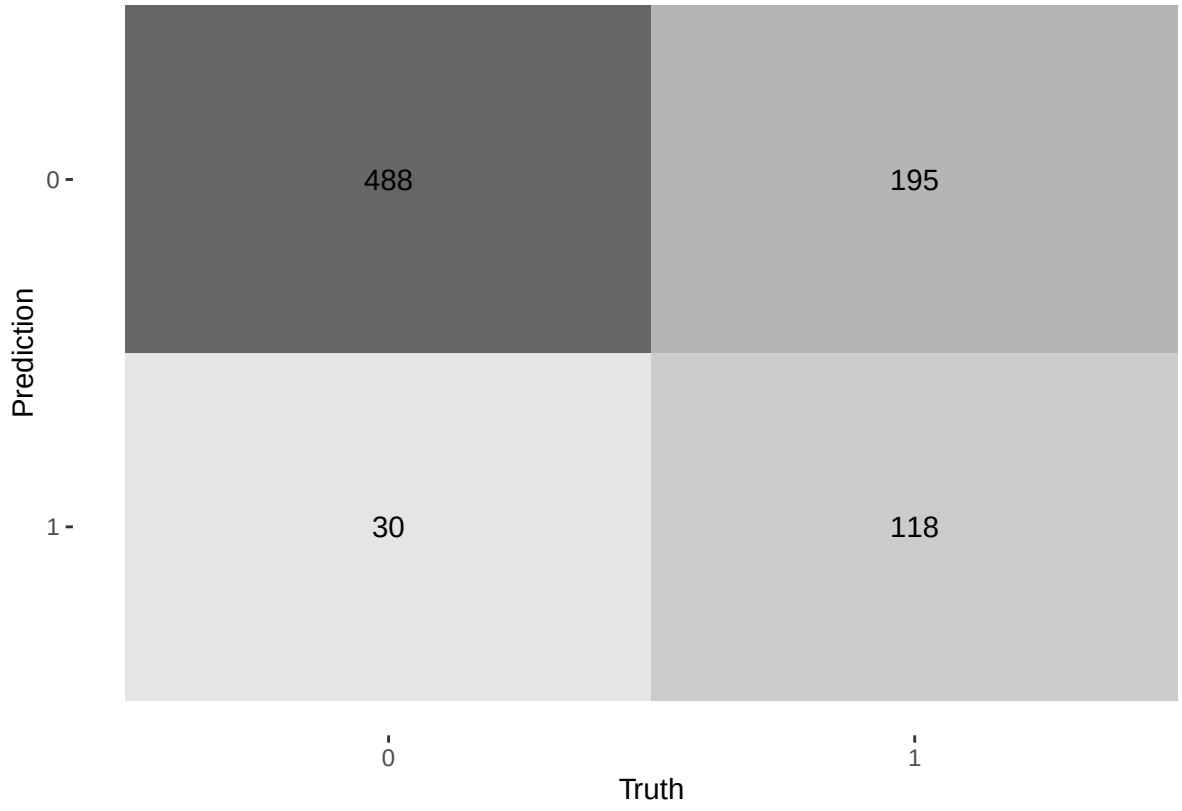
Random forest hash

```
last_fit_rf_hash <- last_fit(workflow_rf_hash,
                        split = tidy_split,
                        metrics = metric_set(
                          recall, precision, f_meas,
                          accuracy, kap,
                          roc_auc, sens, spec)
                        )
```

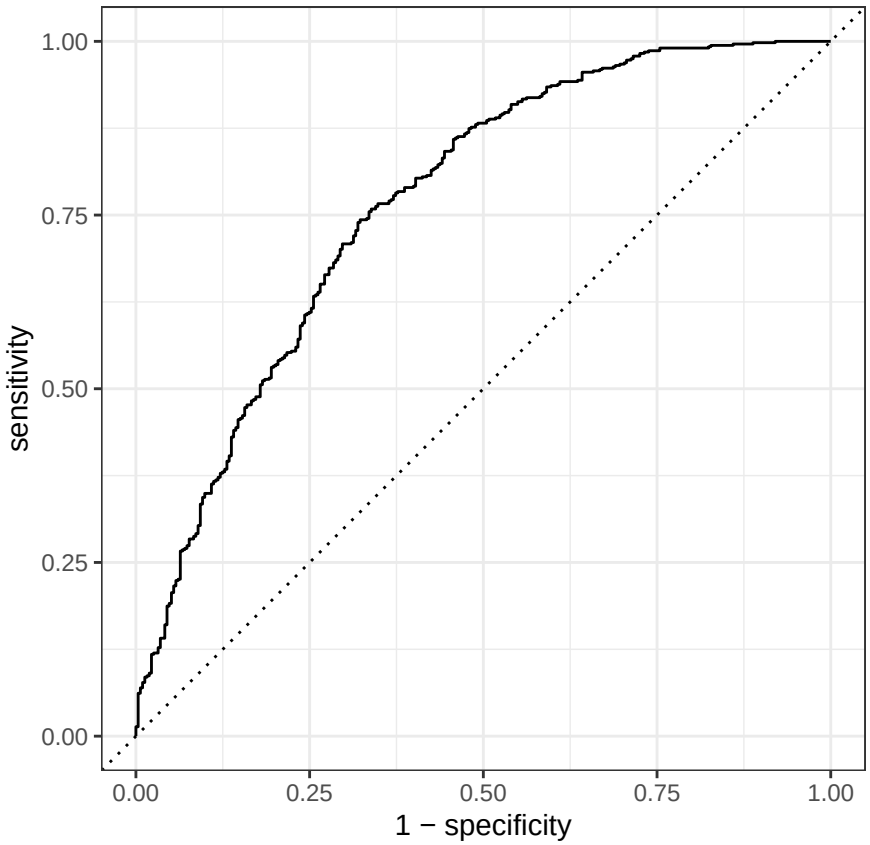
```
last_fit_rf_hash %>%
  collect_metrics()
```

```
## # A tibble: 8 x 4
##   .metric   .estimator .estimate .config
##   <chr>     <chr>          <dbl> <chr>
## 1 recall    binary         0.942 Preprocessor1_Model1
## 2 precision binary         0.714 Preprocessor1_Model1
## 3 f_meas    binary         0.813 Preprocessor1_Model1
## 4 accuracy  binary         0.729 Preprocessor1_Model1
## 5 kap       binary         0.356 Preprocessor1_Model1
## 6 sens      binary         0.942 Preprocessor1_Model1
## 7 spec      binary         0.377 Preprocessor1_Model1
## 8 roc_auc   binary         0.768 Preprocessor1_Model1
```

```
last_fit_rf_hash %>%
  collect_predictions() %>%
  conf_mat(y, .pred_class) %>%
  autoplot(type = "heatmap")
```




```
last_fit_rf_hash %>%
  collect_predictions() %>%
  roc_curve(y, .pred_0) %>%
  autoplot()
```


1.00
0.75
0.50
0.25
0.00
sensitivity
0.00
0.25
0.50
0.75
1.00
1 − specificity