HW6

Alvaro Quijano

1. Explaining the python code

a) Explain this code to me

1. Import all the required libraries.

```
import os
import time
import random
import hashlib
import requests
from bs4 import BeautifulSoup as BS
import string
import nltk
from nltk.corpus import stopwords
from collections import Counter
import csv
```

2. Download the stopwords and the model punkt (tokenizer) and set the stopwords for using with english language text.

```
# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('stopwords')

# Set of English stopwords
stop_words = set(stopwords.words('english'))
```

3. Creates a folder called page_cache but firs check if it exists.

```
CACHE_DIR = "page_cache"

# Create the cache directory if it doesn't exist
if not os.path.exists(CACHE_DIR):
  os.makedirs(CACHE_DIR)
```

4. Generate an hashed URL to use as a unique cache filename.

```
def md5_hash(url):
  """Returns the MD5 hash of a given URL."""
return hashlib.md5(url.encode()).hexdigest()
```

5. Build a path for the cache file with the hashed URL.

```
def cache_path(url):
  """Returns the cache file path for a given URL."""
return os.path.join(CACHE_DIR, md5_hash(url))
```

6. Get the page's content from the web with some difference in time (delay) to avoid overloading the server.

```
def fetch_raw(url):
  """Fetches the page content from the web without caching."""
headers = {
  "User-Agent": "curl/7.68.0",  # Mimic the curl request
  "Accept-Language": "en-US,en;q=0.5"
}

# Print the current time in HH:MM:SS AM/PM format before each fetch
print(f"Fetching {url} at {time.strftime('%I:%M:%S %p')}")

try:
  time.sleep(random.uniform(6, 12))  # Random delay to avoid hammering the server
response = requests.get(url, headers=headers)
if response.status_code == 200:
  return response.text
else:
  print(f"Failed to fetch {url} with status code {response.status_code}")
return None
except requests.RequestException as e:
  print(f"Error fetching {url}: {e}")
return None
```

7. Load the page from cache if it exists; otherwise, get it from the web and caches it.

```
def fetch(url):
  """Wrapper function that implements caching around the raw fetch."""
cache_file = cache_path(url)

# Check if the page is cached
if os.path.exists(cache_file):
  with open(cache_file, 'r', encoding='utf-8') as file:
  print(f"Loading cached page for {url}")
return BS(file.read(), "html.parser")

# If not cached, fetch the raw page and cache it
page_content = fetch_raw(url)
if page_content:
  with open(cache_file, 'w', encoding='utf-8') as file:
  file.write(page_content)
return BS(page_content, "html.parser")
else:
  return None
```

8. Retrieves all episode URLs from the main episode list page.

```
def episode_list_urls():
  """Fetches URLs of episodes from the main episode list page."""
source_url = "http://www.chakoteya.net/NextGen/episodes.htm"
```

```
bs = fetch(source_url)
urls = []
for tb in bs.find_all("tbody"):
  for anchor in tb.find_all("a"):
  urls.append(f"http://www.chakoteya.net/NextGen/{anchor.attrs['href']}")
return urls
```

9. Tokenize, remove punctuation and stopwords, and counts word frequencies.

```
def tokenize_and_count(text):
  """Tokenizes the text, removes punctuation, stopwords, downcases, and counts word frequencies."""
# Remove punctuation and downcase
text = text.translate(str.maketrans("", "", string.punctuation)).lower()

# Tokenize the text
tokens = nltk.word_tokenize(text)

# Remove stop words
filtered_tokens = [word for word in tokens if word not in stop_words]

# Count the word frequencies
word_counts = Counter(filtered_tokens)

return word_counts
```

10. Collect and store the URLs and extracted text for all the episodes.

```
def get_text_of_episodes():
  """Fetches and returns an array of objects with episode URLs and their text."""
urls = episode_list_urls()
episodes = []

for url in urls:
  bs = fetch(url)
b = bs.find("tbody")
txt = b.text

# Store the URL and text in an object (dictionary) for each episode
episodes.append({
  "url": url,
  "text": txt
})

return episodes
```

11. Count word frequencies for each episode and save them by URL.

```
def get_word_counts_for_episodes(episodes):
  """Takes an array of episode objects and calculates word frequencies for each."""
episode_word_counts = {}

for episode in episodes:
  url = episode["url"]
text = episode["text"]
```

```
# Tokenize the text and count word frequencies
word_counts = tokenize_and_count(text)

# Store the word counts for each episode
episode_word_counts[url] = word_counts

return episode_word_counts
```

12. Sums word counts get total frequencies across all episodes.

```
def get_total_word_count(episode_word_counts):
  """Calculates the total word count across all episodes."""
total_word_count = Counter()

for word_counts in episode_word_counts.values():
  total_word_count.update(word_counts)

return total_word_count
```

13. Transforms word counts for each episode into a vector for specific words.

```
def convert_to_word_count_vectors(episode_word_counts, filtered_words):
  """Converts word counts for each episode into a vector following the filtered word order."""
word_vectors = {}

for url, word_counts in episode_word_counts.items():
  # Create a vector for this episode by the order of filtered_words
  vector = [word_counts.get(word, 0) for word in filtered_words]
word_vectors[url] = vector

return word_vectors
```

14. Saves the word count vectors for each episode in a CSV file.

```
def write_word_counts_to_csv(word_count_vectors, filtered_words, filename="episode_word_counts.csv"):
  """Writes the episode word count vectors to a CSV file."""
with open(filename, 'w', newline='', encoding='utf-8') as csvfile:
  writer = csv.writer(csvfile)

# Write the header row (episode URL and each word)
header = ['Episode URL'] + filtered_words
writer.writerow(header)

# Write each episode's word count vector
for url, vector in word_count_vectors.items():
  row = [url] + vector
writer.writerow(row)
```

Finally, words are obtained using the functions described

```
# Example usage to fetch episode texts, get word counts, and calculate total word count
episodes = get_text_of_episodes()
episode_word_counts = get_word_counts_for_episodes(episodes)

# Calculate total word count over all episodes
```

```python
total_word_count = get_total_word_count(episode_word_counts)

# Filter words with a total count greater than 20 and sort by frequency
filtered_words = [word for word, count in total_word_count.items() if count > 20]

# Convert each episode's word counts into a vector of word counts
word_count_vectors = convert_to_word_count_vectors(episode_word_counts, filtered_words)

# Write the word count vectors to a CSV file
write_word_counts_to_csv(word_count_vectors, filtered_words)

print(f"Word counts written to 'episode_word_counts.csv'")
```

b) and c) Configure you docker container and run this code

The docker container is configured using the following code

```
FROM python:3

# Set the working directory inside the container
WORKDIR /usr/src/app

# Copy all files into the container
COPY . .

# Install necessary libraries
RUN pip install requests beautifulsoup4 nltk

# Run the script
CMD ["python", "script.py"]
```

We build the container using the following expression:

```
docker build -t my_python .
```

Then, we run the container pointing to the local folder, as follows:

```
docker run -v "$(pwd)":/usr/src/app my_python
```

2. Visualize this data by

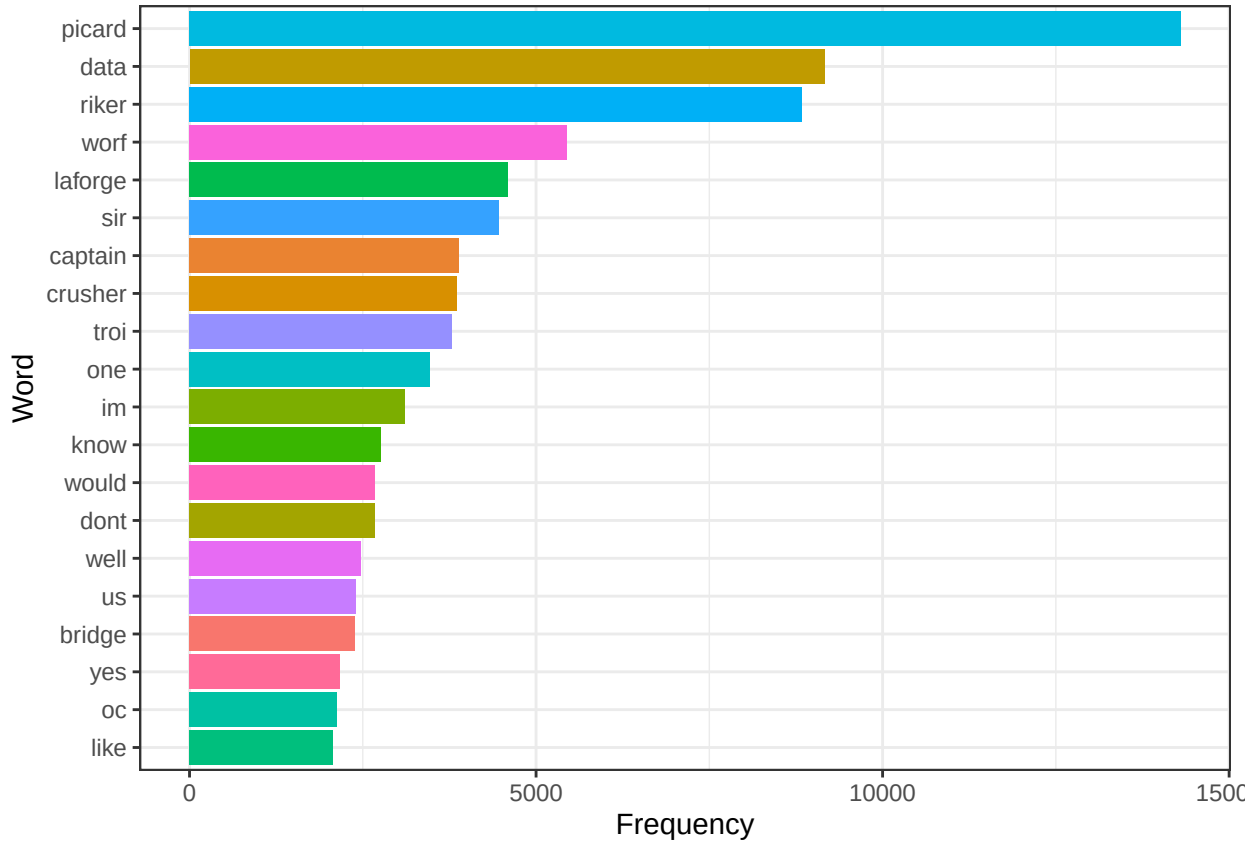
We load the libraries and the dataset,

```r
library(tidyverse)
library(ggplot2)
library(gbm)
library(pROC)

data0 <- read_csv("episode_word_counts.csv")
data <- data0 %>%
        select(-`Episode URL`)
```

a. Showing a histogram of the word counts over the entire ouvre from largest to smallest or to some appropriate small number

We counted the words and the following is the horizontal bar with the 20 largest frequencies in the data.

```
word_counts <- data.frame(`Var` = colnames(data) ,
                          `Freq`= data  %>%
                            colSums())

ggplot(word_counts %>% arrange(-Freq) %>% head(20),
       mapping = aes(y= reorder(Var, Freq), x=Freq, fill = Var)) +
  geom_bar(stat = "identity", show.legend = F) +
  theme_bw() +
  labs(x = "Frequency", y = "Word")
```



We can see that the 3 more frequent words are Picard, Data and Riker which corresponds to the 3 iconic characters from Star Trek.

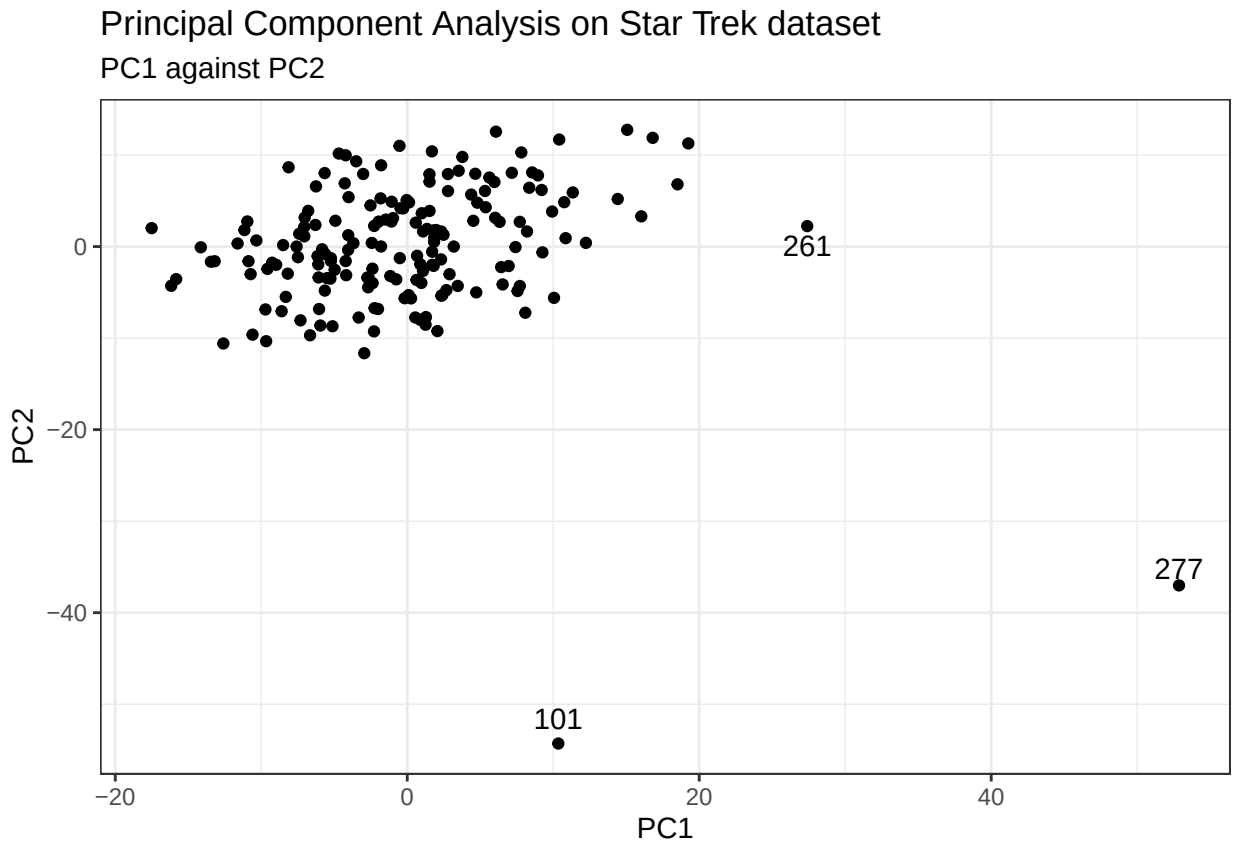
b. Plot reduced dimensionality version of this data set (eg, PCA)

We obtain the Principal Component Analysis (PCA) for the Start Trek dataset as follows:

```
## Running the
pca.res = prcomp(data, center = T, scale. = T)
names = colnames(data)
```

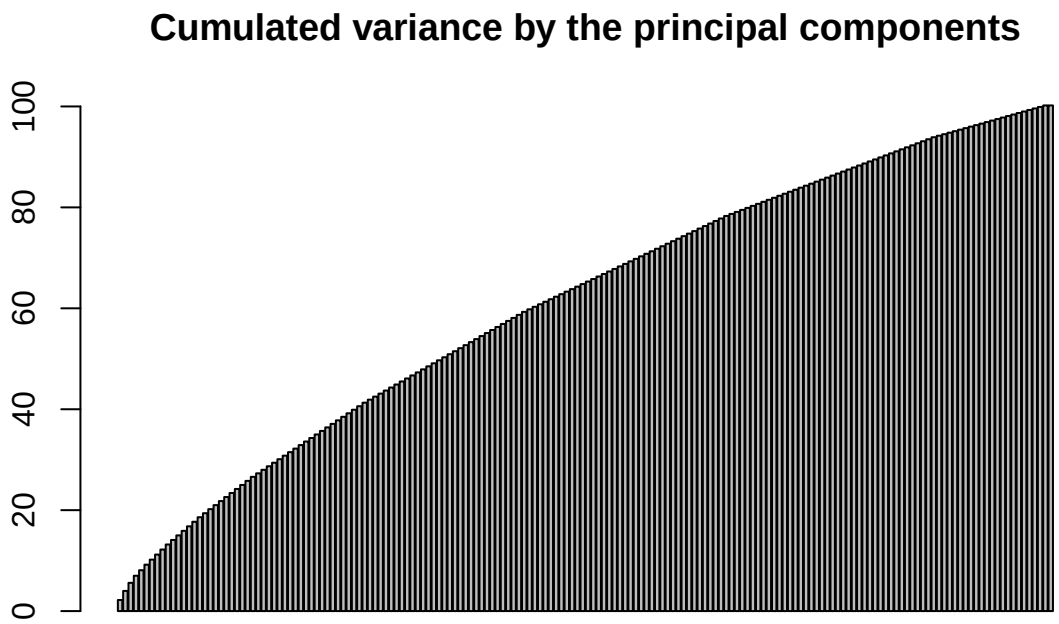
and then plot the PCA reduced data set as

```
db.pca = pca.res$x %>% as.data.frame()
db.pca$episode = substr(data0$`Episode URL`,34,36)

ggplot(db.pca %>% mutate(episode = ifelse(episode %in% c("261","277","101"), episode, "")),
          mapping=aes(x = PC1, y = PC2, label=episode)) +
      geom_point() + geom_text(position = position_stack(vjust = 0.05)) +
      theme_bw() +
      labs(title ="Principal Component Analysis on Star Trek dataset", subtitle ="PC1 against PC2")
```



Note that 3 episodes (101, 261 and 277) have extreme values for both PC1 and PC2. Episodes 101 and 277 correspond to the first and last episodes extracted.

```
## Getting the explained variance
expl.var <- round(pca.res$sdev^2/sum(pca.res$sdev^2)*100,1) # percent explained variance

##
barplot(cumsum(expl.var),
        main = "Cumulated variance by the principal components", width = 0.6)
```




```
###
cat("Cumulated variance for the first 10 components")
```

```
## Cumulated variance for the first 10 components
```

```
cumsum(expl.var)[1:10]
```

```
##  [1]  2.2  4.0  5.6  7.0  8.1  9.2 10.2 11.2 12.2 13.2
```

According to the accumulated variance, a significant number of components are needed to explain the variability in the dataset. Observe that the first 10 components explain only 13% of the variation in the dataset.

c. Color code the data points by finding for each vector the name of the character that appears most frequently (picard, riker, data, troi, worf, crusher) and provide insightful comments if possible.

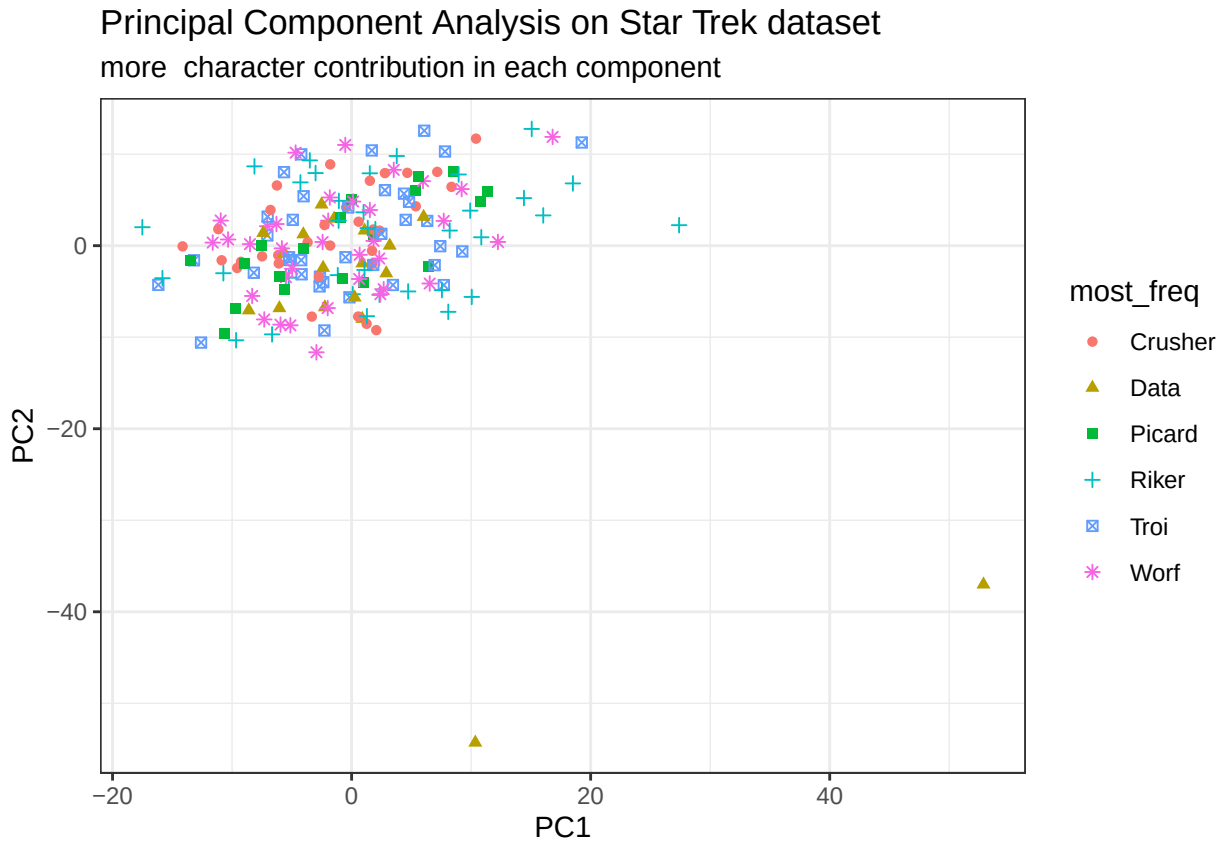
For this, we need to check the loadings,

```
## Using loadings identifying the more frequent character in each component
most_frequent = pca.res$rotation %>%
    as_tibble() %>%
    mutate(word=colnames(data)) %>%
    filter(word %in% c("picard", "riker", "data", "troi", "worf", "crusher")) %>%
    select(-word) %>%
    mutate(across(everything(), abs)) %>%
    summarise(across(everything(), ~ which.max(.))) %>% t() %>%
```

```
    as_tibble() %>%
    mutate(m_freq = case_when(V1==1 ~ 'Riker',
                              V1==2 ~ 'Picard',
                              V1==3 ~ 'Data',
                              V1==4 ~ 'Troi',
                              V1==5 ~ 'Worf',
                              V1==6 ~ 'Crusher')) %>%
  pull(m_freq)
db.pca$most_freq = most_frequent
```

Thus, we plot the news components by coloring each episode with the most frequent character

```
ggplot(db.pca,
          mapping=aes(x = PC1, y = PC2, color = most_freq, shape = most_freq)) +
      geom_point() +
      theme_bw() +
      labs(title ="Principal Component Analysis on Star Trek dataset", subtitle ="more  character contri
```

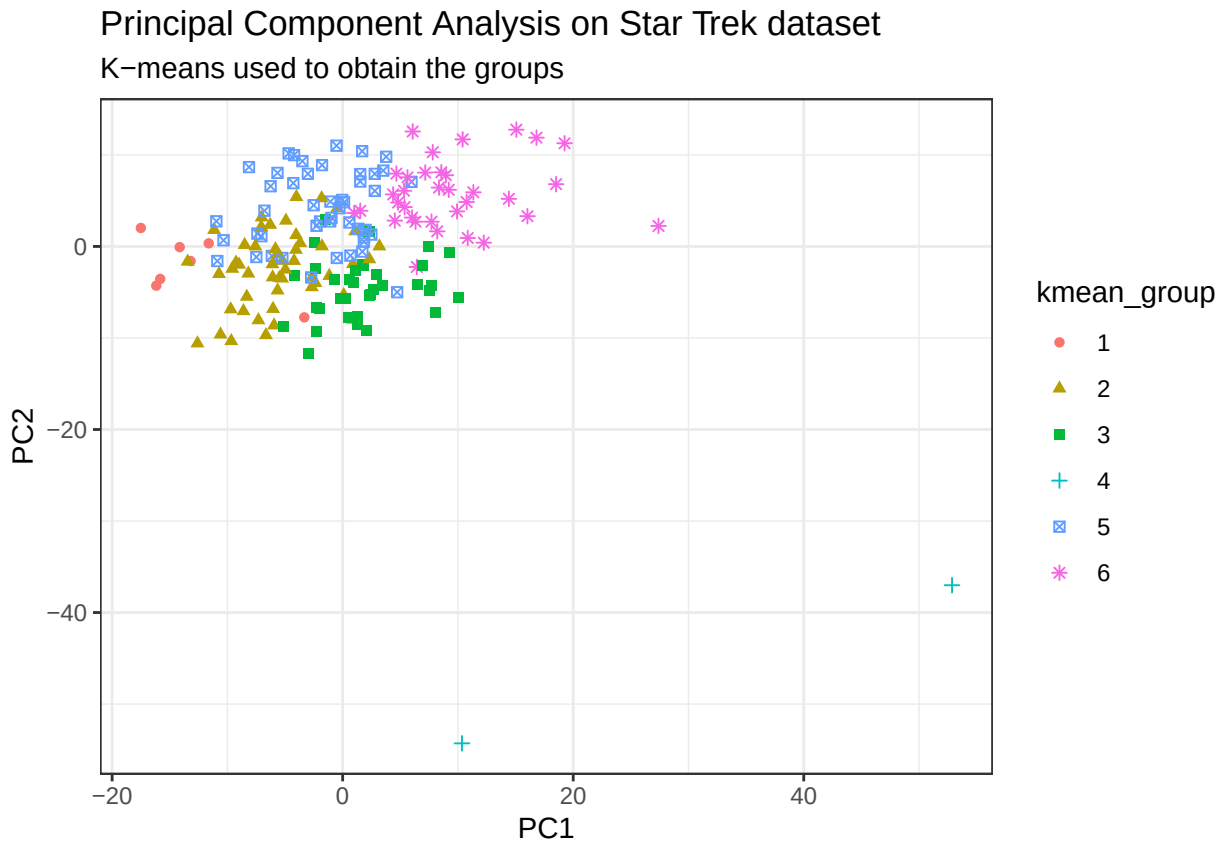


Note that PC1 and PC2 can help distinguish observations related to the character Data (green triangle), and also Riker (plus sign in light blue).

3. Cluster the data set. Maybe choose a number of clusters equal to the above list of characters. Visualize the results by color coding the 2d data from problem 2 on a scatter plot.

We use k-means to obtain the grouping for each episode

```
db.pca$kmean_group = factor((kmeans(db.pca %>% select(-c(episode,most_freq)), centers = 6))$cluster)

ggplot(db.pca,
          mapping=aes(x = PC1, y = PC2, color = kmean_group, shape = kmean_group)) +
      geom_point() +
      theme_bw() +
      labs(title ="Principal Component Analysis on Star Trek dataset", subtitle ="K-means used to obtai
```



Using k means we obtained two different groups for the 2 outliers values. Overall is a good classification that differs from

4. Use adaboost (gbm) to attempt to build a classifier that detects the season of the episode. Describe which words are most useful for determining the season. How well does the classifier work? For simplicity's sake, make another classifier which predicts whether the episode is from the first or second half of the show's run. Plot the ROC curve and give the Area under said.

For this exercise, we exclude the columns (words) that have a frequency of less than 130 (arbitrary selection) in all episodes (less than 1 count/episode) to avoid any problem with variance when running the GBM algorithm.

```
data_model = data %>%  select(
col_sums_df <- tibble(
  variable = names(data),
  total = colSums(data)
) %>% filter(total> 130) %>%
      pull(variable)) %>%
      mutate(first_half = (row_number() < max(row_number()) / 2) * 1)
```

Split the data in train and set datasets,

```
train_ii <- runif(nrow(data_model)) < 0.75
train <- data_model %>% filter(train_ii)
test <- data_model %>% filter(!train_ii)
```

And finally, run the GMB algorithm and get the predicted values

```
model <- gbm(first_half ~ ., data = train)
```

```
## Distribution not specified, assuming bernoulli ...
```

```
## the probability that it corresponds to the first 2 seasons
predicted_values <- predict(model, newdata = test, type = "response")
```

```
## Using 100 trees...
```

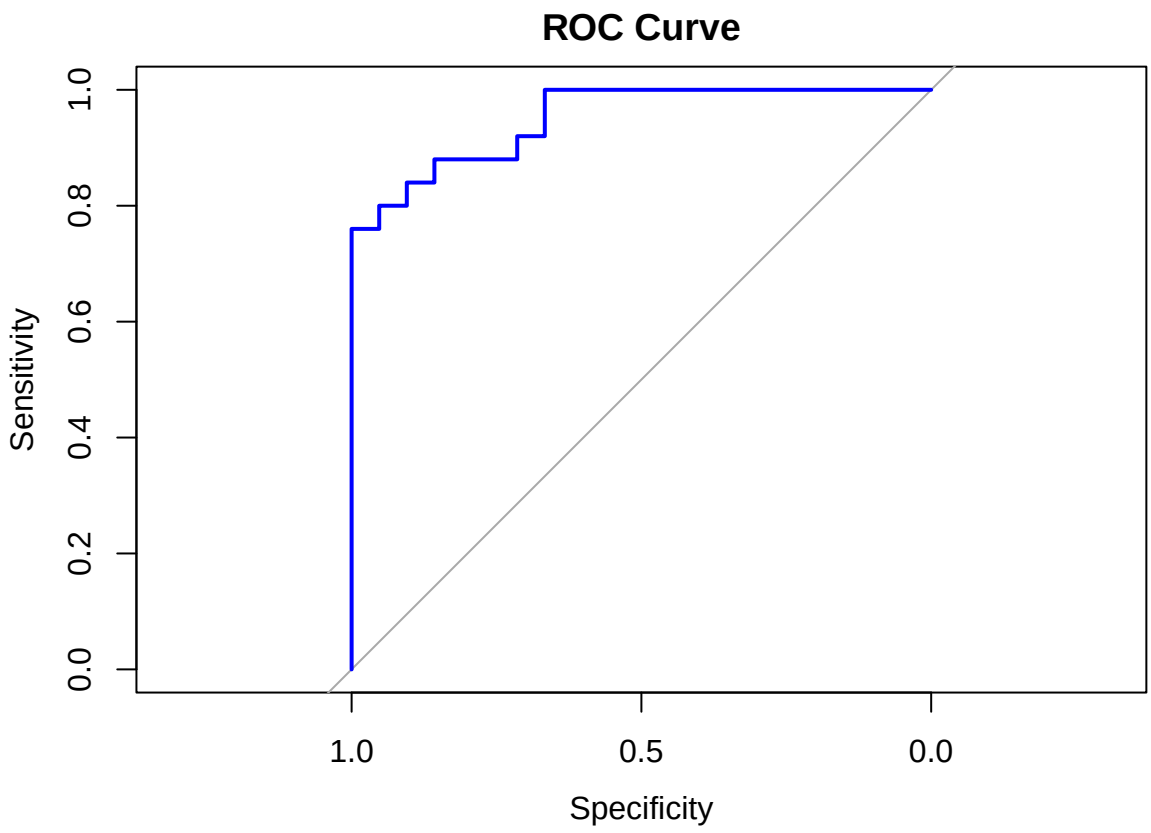
Finally we obtain the ROC curve and the Area Under the Curve

```
# Load necessary library

roc_curve <- roc(test$first_half, predicted_values)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

```
# Plot the ROC curve
plot(roc_curve, main = "ROC Curve", col = "blue")
```

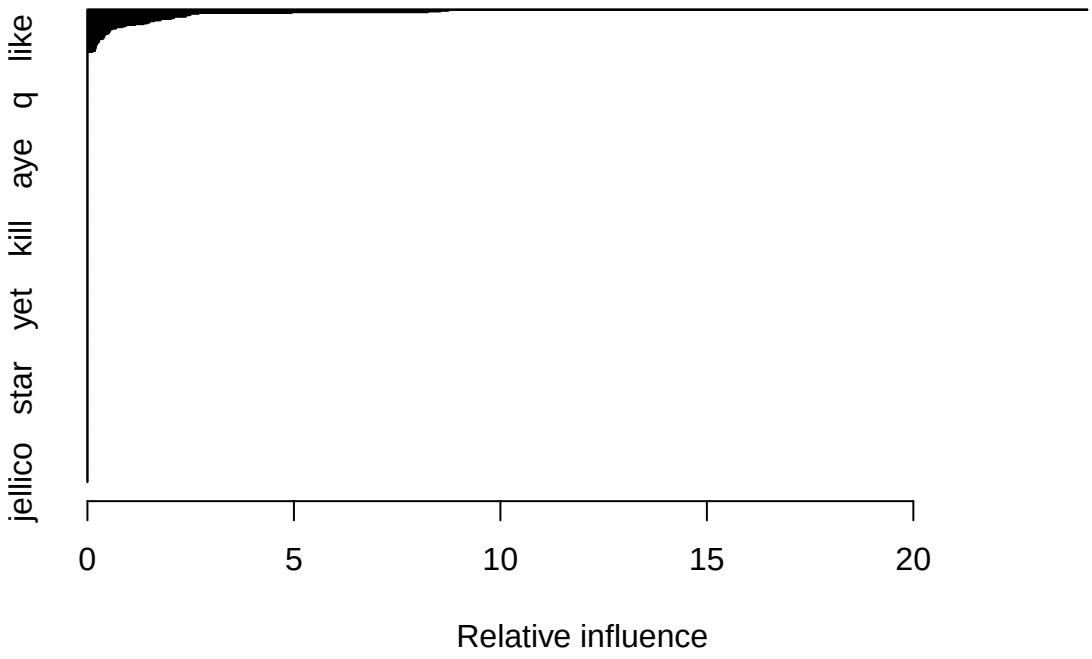



```
# Optionally, print the AUC (Area Under the Curve)
auc_value <- auc(roc_curve)
print(paste("AUC:", round(auc_value,4)))
```

```
## [1] "AUC: 0.9505"
```

Note that we have a very good classifier with AUC: 0.9828. Now we examine the contribution to the first and second half,

```
summary(model) %>% head(5)
```




```
##                     var   rel.inf
## wesley           wesley 24.213216
## number           number  8.743583
## found             found  8.549406
## plasma           plasma  8.245899
## transporter transporter  4.948668
```

Observe that Wesley is a strong presence in the first half of Star Trek: The Next Generation, as he appears as a regular character in 4 seasons. The term 'Number' likely refers to 'Number One,' which is how Captain Picard addresses Riker, his trusted First Officer.

5. Load the data into a pandas data frame. Count the rows. Calculate the row sum for the data set's numerical columns. Remove rows with less than 100 total counts. Write the data back out.

I decided to do the column sum because there is no filtering in the dataset when using the row sum. The following is the python code employed:

```
import pandas as pd

# Load the data
data = pd.read_csv("episode_word_counts.csv")  # Replace with your actual file path

# Obtaining the number of columns
print(f"The number of columns in the dataset is: {data.shape[1]}")
```

```
# Calculate the column sum for numerical columns
column_sums = data.select_dtypes(include=['number']).sum()

# Define the columns that we would keep in the dataset
keep_cols = column_sums[column_sums >= 100].index

# Get the filtered data
filtered_data = data[keep_cols]
print(f"The number of columns after filtering the dataset is: {filtered_data.shape[1]}")

# Write to a CSV file
filtered_data.to_csv("filtered_data.csv", index=False)
```