HW1_spa9659

February 28, 2024

#Homework 1 - Text as Data

Name: Sampreeth Avvari

NetID:: spa9659

#Import statements

[7]:
```
from google.colab import files

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

import os
import string

import requests
import random
import re

import matplotlib.pyplot as plt

import nltk
import string
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer, PorterStemmer
from nltk.corpus import stopwords
from nltk.collocations import BigramCollocationFinder
from nltk.metrics import BigramAssocMeasures
from scipy.stats import chi2, chi2_contingency

from tqdm import tqdm

from pathlib import Path
from nltk.tokenize import word_tokenize
import string
```

```
import os
import matplotlib.pyplot as plt

nltk.download('wordnet')
nltk.download('punkt')
nltk.download('stopwords')
!pip install textstat
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data…
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Unzipping corpora/stopwords.zip.
```

```
Collecting textstat
  Downloading textstat-0.7.3-py3-none-any.whl (105 kB)
                           105.1/105.1

kB 3.1 MB/s eta 0:00:00
Collecting pyphen (from textstat)
  Downloading pyphen-0.14.0-py3-none-any.whl (2.0 MB)
                           2.0/2.0 MB
12.9 MB/s eta 0:00:00
Installing collected packages: pyphen, textstat
Successfully installed pyphen-0.14.0 textstat-0.7.3
```

[8]:
```
!pip install py-readability-metrics
from readability import Readability
```

```
Collecting py-readability-metrics
  Downloading py_readability_metrics-1.4.5-py3-none-any.whl (26 kB)
Requirement already satisfied: nltk in /usr/local/lib/python3.10/dist-packages
(from py-readability-metrics) (3.8.1)
Requirement already satisfied: click in /usr/local/lib/python3.10/dist-packages
(from nltk->py-readability-metrics) (8.1.7)
Requirement already satisfied: joblib in /usr/local/lib/python3.10/dist-packages
(from nltk->py-readability-metrics) (1.3.2)
Requirement already satisfied: regex>=2021.8.3 in
/usr/local/lib/python3.10/dist-packages (from nltk->py-readability-metrics)
(2023.12.25)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages
(from nltk->py-readability-metrics) (4.66.2)
Installing collected packages: py-readability-metrics
Successfully installed py-readability-metrics-1.4.5
```

[106]:
```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving 1981-Reagan.txt to 1981-Reagan.txt
Saving 1985-Reagan.txt to 1985-Reagan.txt
```

#Question 1

[107]:
```python
with open('1981-Reagan.txt', 'r') as file:
  q1_r_1981 = file.read()

with open('1985-Reagan.txt', 'r') as file:
  q1_r_1985 = file.read()
```

[108]:
```python
def q1_preprocess(doc):
  d = doc.translate(str.maketrans('', '', string.punctuation))
  tokens = word_tokenize(d)
  return ' '.join(tokens)
```

##1 A

[109]:
```python
def q1_pre(doc):
  d = doc.translate(str.maketrans('', '', string.punctuation))
  tokens = word_tokenize(d)
  return tokens
```

[110]:
```python
len(q1_r_1981)
```

[110]: 13744

[112]:
```python
q1_tokens_1981 = q1_pre(q1_r_1981)
```

[120]:
```python
q1_types_1981 = len(set(q1_tokens_1981))  # Identifying number of unique tokens␣
 ↪aka types
q1_total_tokens_1981 = len(q1_tokens_1981)  # Total number of tokens
q1_ttr_1981 = q1_types_1981 / q1_total_tokens_1981  # TTR
print("The TTR for the Reagan 1981 speech is {}".format(q1_ttr_1981))
q1_tokens_1985 = q1_pre(q1_r_1985)  # Tokenizing the 1985 speech
q1_types_1985 = len(set(q1_tokens_1985))  # Identifying number of unique tokens␣
 ↪aka types
q1_total_tokens_1985 = len(q1_tokens_1985)  # Total number of tokens
q1_ttr_1985 = q1_types_1985 / q1_total_tokens_1985  # TTR
print("The TTR for Reagan's 1985 speech is {}".format(q1_ttr_1985))
q1_G_1981 = q1_types_1981 / np.sqrt(q1_total_tokens_1981)
print("The Guiraud’s index of Reagan 1981 speech is {}".format(q1_G_1981))
q1_G_1985 = q1_types_1985 / np.sqrt(q1_total_tokens_1985)
print("The Guiraud’s index of Reagan 1985 speech is {}".format(q1_G_1985))
```

```
The TTR for the Reagan 1981 speech is 0.36800986842105265
The TTR for Reagan's 1985 speech is 0.35608424336973477
The Guiraud’s index of Reagan 1981 speech is 18.14852148900406
The Guiraud’s index of Reagan 1985 speech is 18.03066593879904
```

The type-token ratio (TTR) of the 1985 speech is slightly lower than that of the 1981 speech. This suggests that the 1981 speech might be more lexically diverse compared to 1985. However, it's important to note that we haven't performed additional preprocessing steps such as removing stopwords or stemming, so we cannot conclusively determine the lexical richness based solely on this observation.

##1 B

[119]:
```
q1_reagen = [q1_r_1981, q1_r_1985]
q1_r_preproc = list(map(q1_preprocess, q1_reagen))
q1_vectorizer = CountVectorizer()
q1_dtm = q1_vectorizer.fit_transform(q1_r_preproc)
np.unique(q1_dtm[0].toarray())
from sklearn.metrics.pairwise import cosine_similarity
q1_cos = cosine_similarity(q1_dtm[0], q1_dtm[1])
print("The cosine similarity between the 2 speeches is {}".format(q1_cos[0][0]))
```

```
The cosine similarity between the 2 speeches is 0.959248178749997
```

The high cosine similarity value, which is close to 1, indicates that the speeches are very similar to each other.

#Question 2

##2A

a) Stemming:

- The TTR should decrease because stemming reduces words to their root form, leading to fewer unique word forms.
- In terms of similarity, it might increase because different word forms are now considered the same.

[121]:
```
def q2_preprocess_2a(doc):
  d = doc.translate(str.maketrans('', '', string.punctuation))
  tokens = word_tokenize(d)
  output = [PorterStemmer().stem(word) for word in tokens]

  return output
```

[123]:
```
q2_r_1981_pre = q2_preprocess_2a(q1_r_1981)
q2_uniq_1981 = len(set(q2_r_1981_pre))
q2_total_tokens_1981_2a = len(q2_r_1981_pre)
q2_ttr_1981_2a = q2_uniq_1981 / q2_total_tokens_1981_2a
print("The TTR of Reagan's 1981 speech with stemming is {}".
 ↪format(q2_ttr_1981_2a))

q2_r_1985_pre = q2_preprocess_2a(q1_r_1985)
q2_uniq_1985 = len(set(q2_r_1985_pre))
q2_total_tokens_1985_2a = len(q2_r_1985_pre)
q2_ttr_1985_2a = q2_uniq_1985 / q2_total_tokens_1985_2a
```

```
print("The TTR of Reagan's 1985 speech with stemming is {}".
  ↪format(q2_ttr_1985_2a))

# Guiraud
q2_g_1981_2a = q2_uniq_1981 / np.sqrt(q2_total_tokens_1981_2a)
print("The Guiraud’s index of Reagan 1981 speech with stemming is {}".
  ↪format(q2_g_1981_2a))

q2_g_1985_2a = q2_uniq_1985 / np.sqrt(q2_total_tokens_1985_2a)
print("The Guiraud’s index of Reagan 1985 speech with stemming is {}".
  ↪format(q2_g_1985_2a))
```

```
The TTR of Reagan's 1981 speech with stemming is 0.3079769736842105
The TTR of Reagan's 1985 speech with stemming is 0.297191887675507
The Guiraud’s index of Reagan 1981 speech with stemming is 15.187980553367643
The Guiraud’s index of Reagan 1985 speech with stemming is 15.048595230410589
```

[124]:
```
def q2_preprocess_2a_dtm(doc):
  d = doc.translate(str.maketrans('', '', string.punctuation))
  tokens = word_tokenize(d)
  output = [PorterStemmer().stem(word) for word in tokens]

  return ' '.join(output)

# Preprocessing
q2_r_preproc_2a_dtm = list(map(q2_preprocess_2a_dtm, q1_reagen))
q2_vectorizer_2a = CountVectorizer()
q2_dtm_2a = q2_vectorizer_2a.fit_transform(q2_r_preproc_2a_dtm)

q2_cos_2a = cosine_similarity(q2_dtm_2a[0], q2_dtm_2a[1])
print("The cosine similarity between the 2 speeches with stemming is {}".
  ↪format(q2_cos_2a[0][0]))
```

```
The cosine similarity between the 2 speeches with stemming is 0.9604945711581567
```

##2 B

b) Removing Stop Words:

- The TTR might increase because removing stop words reduces the total number of unique words, leading to a higher ratio of unique words to total words.
- In terms of similarity, it might decrease because removing common stop words focuses the comparison on the remaining content words, potentially highlighting more meaningful differences.

[125]:
```
def q2_preprocess_2b(doc):
  d = doc.translate(str.maketrans('', '', string.punctuation))
  tokens = word_tokenize(d)
  stop_words = set(stopwords.words("english"))
```

```
    output_2b = [word for word in tokens if word not in stop_words]

    return output_2b

q2_r_1981_pre_2b = q2_preprocess_2b(q1_r_1981)
q2_uniq_1981_2b = len(set(q2_r_1981_pre_2b))
q2_total_tokens_1981_2b = len(q2_r_1981_pre_2b)
q2_ttr_1981_2b = q2_uniq_1981_2b / q2_total_tokens_1981_2b
print("The TTR of Reagan's 1981 speech after removing stop words is {}".
 ↪format(q2_ttr_1981_2b))

q2_r_1985_pre_2b = q2_preprocess_2b(q1_r_1985)
q2_uniq_1985_2b = len(set(q2_r_1985_pre_2b))
q2_total_tokens_1985_2b = len(q2_r_1985_pre_2b)
q2_ttr_1985_2b = q2_uniq_1985_2b / q2_total_tokens_1985_2b
print("The TTR of Reagan's 1985 speech after removing stop words is {}".
 ↪format(q2_ttr_1985_2b))

# Guiraud
q2_g_1981_2b = q2_uniq_1981_2b / np.sqrt(q2_total_tokens_1981_2b)
print("The Guiraud’s index of Reagan 1981 speech after removing stop words is␣
 ↪{}".format(q2_g_1981_2b))

q2_g_1985_2b = q2_uniq_1985_2b / np.sqrt(q2_total_tokens_1985_2b)
print("The Guiraud’s index of Reagan 1985 speech after removing stop words is␣
 ↪{}".format(q2_g_1985_2b))
```

```
The TTR of Reagan's 1981 speech after removing stop words is 0.6290449881610103
The TTR of Reagan's 1985 speech after removing stop words is 0.5828002842928216
The Guiraud’s index of Reagan 1981 speech after removing stop words is
22.39082078808915
The Guiraud’s index of Reagan 1985 speech after removing stop words is
21.860837886963843
```

[126]:
```
def q2_preprocess_2b_dtm(doc):
    d = doc.translate(str.maketrans('', '', string.punctuation))
    tokens = word_tokenize(d)
    stop_words = set(stopwords.words("english"))
    output_2b = [word for word in tokens if word not in stop_words]
    return ' '.join(output_2b)

# 2b Preprocessing
q2_r_preproc_2b_dtm = list(map(q2_preprocess_2b_dtm, q1_reagen))
q2_vectorizer_2b = CountVectorizer()
q2_dtm_2b = q2_vectorizer_2b.fit_transform(q2_r_preproc_2b_dtm)

q2_cos_2b = cosine_similarity(q2_dtm_2b[0], q2_dtm_2b[1])
```

```
print("The cosine similarity between the 2 speeches after removing stop words␣
↪is {}".format(q2_cos_2b[0][0]))
```

```
The cosine similarity between the 2 speeches after removing stop words is
0.7074650600415002
```

##2 C

c) Converting all words to lowercase:

- The TTR might remain the same or decrease because converting all words to lowercase collapses different case variations of the same word into one, reducing the number of unique word forms.
- In terms of similarity, it might remain the same or increase because different case variations of the same word are now considered identical, potentially increasing the overlap between documents.

[128]:
```
def q2_preprocess_2c(doc):
  d = doc.translate(str.maketrans('', '', string.punctuation))
  tokens = word_tokenize(d)
  output_2c = [word.lower() for word in tokens]
  return output_2c

q2_r_1981_pre_2c = q2_preprocess_2c(q1_r_1981)
q2_uniq_1981_2c = len(set(q2_r_1981_pre_2c))
q2_total_tokens_1981_2c = len(q2_r_1981_pre_2c)
q2_ttr_1981_2c = q2_uniq_1981_2c / q2_total_tokens_1981_2c
print("The TTR of Reagan's 1981 speech after converting words to lowercase is␣
↪{}".format(q2_ttr_1981_2c))

q2_r_1985_pre_2c = q2_preprocess_2c(q1_r_1985)
q2_uniq_1985_2c = len(set(q2_r_1985_pre_2c))
q2_total_tokens_1985_2c = len(q2_r_1985_pre_2c)
q2_ttr_1985_2c = q2_uniq_1985_2c / q2_total_tokens_1985_2c
print("The TTR of Reagan's 1985 speech after converting words to lowercase is␣
↪{}".format(q2_ttr_1985_2c))

# Guiraud 2c
q2_g_1981_2c = q2_uniq_1981_2c / np.sqrt(q2_total_tokens_1981_2c)
print("The Guiraud's index of Reagan 1981 speech after converting words to␣
↪lowercase is {}".format(q2_g_1981_2c))

# Guiraud 2c
q2_g_1985_2c = q2_uniq_1985_2c / np.sqrt(q2_total_tokens_1985_2c)
print("The Guiraud's index of Reagan 1985 speech after converting words to␣
↪lowercase is {}".format(q2_g_1985_2c))
```

```
The TTR of Reagan's 1981 speech after converting words to lowercase is
0.34662828947368424
```

```
The TTR of Reagan's 1985 speech after converting words to lowercase is
0.3369734789391576
The Guiraud’s index of Reagan 1981 speech after converting words to lowercase is
17.094082251654104
The Guiraud’s index of Reagan 1985 speech after converting words to lowercase is
17.062974119520668
```

[129]:
```python
def q2_preprocess_2c_dtm(doc):
  d = doc.translate(str.maketrans('', '', string.punctuation))
  tokens = word_tokenize(d)
  output_2c = [word.lower() for word in tokens]
  return ' '.join(output_2c)

# Preprocessing
q2_r_preproc_2c_dtm = list(map(q2_preprocess_2c_dtm, q1_reagen))
q2_vectorizer_2c = CountVectorizer()
q2_dtm_2c = q2_vectorizer_2c.fit_transform(q2_r_preproc_2c_dtm)

q2_cos_2c = cosine_similarity(q2_dtm_2c[0], q2_dtm_2c[1])
print("The cosine similarity between the 2 speeches after converting words to␣
↪lowercase is {}".format(q2_cos_2c[0][0]))
```

```
The cosine similarity between the 2 speeches after converting words to lowercase
is 0.959248178749997
```

##2 D

d) TF-IDF weighting:

- TF-IDF weighting assigns higher weights to terms that are more important in a document relative to the entire corpus. In this case, with only two documents, the effectiveness of TF-IDF may be limited.
- While TF-IDF can potentially capture the relative importance of terms within each document, the similarity results produced by TF-IDF and simple count vectorization may not differ significantly due to the small size of the corpus.
- Whether TF-IDF makes sense depends on the specific characteristics of the documents and the goals of the analysis. In larger document collections, TF-IDF is generally more effective at capturing the importance of terms. However, in this case, it may not offer significant advantages over simpler methods.

[130]:
```python
def q2_preprocess_2d(doc):
  d = doc.translate(str.maketrans("","", string.punctuation)).lower()
  tokens = word_tokenize(d)
  return " ".join(tokens)

q2_vectorizer_2d = CountVectorizer()
q2_transformer_2d = TfidfTransformer()

q2_reagen_pre_2d = list(map(q2_preprocess_2d, q1_reagen))
```

```
q2_dtm_reagan_2d = q2_vectorizer_2d.fit_transform(q2_reagen_pre_2d)
q2_dtm_reagan_tfidf = q2_transformer_2d.fit_transform(q2_dtm_reagan_2d)

q2_cosine_similarity_count = cosine_similarity(q2_dtm_reagan_2d[0],␣
↪q2_dtm_reagan_2d[1])[0][0]
q2_cosine_similarity_tfidf = cosine_similarity(q2_dtm_reagan_tfidf[0],␣
↪q2_dtm_reagan_tfidf[1])[0][0]

print(f"The Cosine Similarity distance between the speeches using␣
↪countvectorizer is {q2_cosine_similarity_count}")
print(f"The Cosine Similarity distance between the speeches tfidf vectorizer is␣
↪{q2_cosine_similarity_tfidf}")
```

```
The Cosine Similarity distance between the speeches using countvectorizer is
0.959248178749997
The Cosine Similarity distance between the speeches tfidf vectorizer is
0.9454768821402613
```

#Question 3

##3 A

[133]:
```
q3_s1="China Condemns U.S. Decision to Shoot Down Spy Balloon."
q3_s2="U.S. Shoots Down Suspected Chinese Spy Balloon, Recovery Under Way."

def q3_preprocess(doc):
  d = doc.translate(str.maketrans('', '', string.punctuation)).lower()
  tokens = word_tokenize(d)
  return ' '.join(tokens)

q3_l = [q3_s1, q3_s2]

q3_preproc = list(map(q3_preprocess, q3_l))
q3_vectorizer = CountVectorizer()

q3_dtm = q3_vectorizer.fit_transform(q3_preproc)
q3_dtm_a = q3_dtm.toarray()
```

##3 B

[135]:
```
q3_euclidean = np.sqrt(np.sum((q3_dtm_a[0] - q3_dtm_a[1]) ** 2, axis=0))
print("The Euclidean distance between the two sentences is {}".
↪format(q3_euclidean))
```

```
The Euclidean distance between the two sentences is 3.3166247903554
```

##3 C

[136]:
```
q3_manhattan = np.sum(np.abs((q3_dtm_a[0] - q3_dtm_a[1])), axis=0)
```

```
print("The Manhattan distance between the two sentences is {}".
 ↪format(q3_manhattan))
```

```
The Manhattan distance between the two sentences is 11
```

##3 D

[137]:
```
q3_set1 = set(q3_s1)
q3_set2 = set(q3_s2)
q3_jaccard = len(q3_set1.intersection(q3_set2)) / len(q3_set1.union(q3_set2))
print("The Jaccard similarity between the two sentences is {}".
 ↪format(q3_jaccard))
```

```
The Jaccard similarity between the two sentences is 0.75
```

##3 E

[138]:
```
q3_cosine = np.sum((q3_dtm_a[0].dot(q3_dtm_a[1])), axis=0) / (np.sqrt(np.
 ↪sum(q3_dtm_a[0] ** 2, axis=0)) * np.sqrt(np.sum(q3_dtm_a[1] ** 2, axis=0)))
print("The Cosine similarity between the two sentences is {}".format(q3_cosine))
```

```
The Cosine similarity between the two sentences is 0.4216370213557839
```

##3 F

The initial words are “surveillance” and “surveyance.” 1. Change the “i” in “surveillance” to “y” to match “surveyance,” resulting in “surveyllance.” 2. Remove the first “l” in “surveyllance” to match “surveyance,” resulting in “surveylance.” 3. Remove the remaining “l” in “surveylance” to match “surveyance,” resulting in “surveyance.”

Each step represents a point in the process, and the total Levenshtein distance is the sum of these points, which in this case is 3.

[140]:
```
def q3_levenshtein_distance(s1, s2):
    # Initialize matrix of zeros
    rows = len(s1) + 1
    cols = len(s2) + 1
    distance = [[0 for x in range(cols)] for x in range(rows)]

    # Populate matrix of zeros with the indices of each character of both␣
 ↪strings
    for i in range(1, rows):
        distance[i][0] = i
    for j in range(1, cols):
        distance[0][j] = j

    # Iterate over the matrix to compute the cost of deletions, insertions, and␣
 ↪substitutions
    for col in range(1, cols):
        for row in range(1, rows):
            if s1[row - 1] == s2[col - 1]:
```

```
                cost = 0  # If the characters are the same in the two strings␣
↪in a given position [i,j] then the cost is 0
            else:
                cost = 1  # If not, you calculate the cost of a substitution
            distance[row][col] = min(distance[row - 1][col] + 1,  # Cost of␣
↪deletions
                                 distance[row][col - 1] + 1,  # Cost of␣
↪insertions
                                 distance[row - 1][col - 1] + cost)  #␣
↪Cost of substitutions
    return distance[row][col]

# Example usage
q3_s1 = "surveillance"
q3_s2 = "surveyance"
q3_levenshtein_distance(q3_s1, q3_s2)
```

[140]: 3

#Question 4

##4 A

[39]:
```
id = [[84, 4695, 6447, 15238],[70, 74, 76, 86], [2814, 2817, 4217, 4300],␣
↪[2868, 4223, 4531, 4946]]
book_list = []
for i in id:
  auth = ""
  for j in i:
    lines = requests.get(r"https://www.gutenberg.org/cache/epub/"+ str(j) +"/
↪pg"+ str(j) +".txt")
    temp = ""
    for i in range(500):
      temp += str(random.choice(list(lines)[100:]))
    auth += temp
  book_list.append(auth)
```

##4 B

[54]:
```
stop_words = set(stopwords.words("english"))
def q4_preprocess(text):
    #observed these unwanted characters
    escapes = {
          '\\a': ' ', '\\b': ' ', '\\f': ' ', '\\n': ' ',
          '\\r': ' ', '\\t': ' ', '\\v': ' ', "\\'": ' ',
          '\\"': ' ', '\\\\': ' ', '\\?':" " ,"b\'":" "
      }
    for escape, replacement in escapes.items():
```

```
        text = text.replace(escape, replacement)
    text = re.sub(r'\\x[0-9A-Fa-f]{2}', ' ', text)
    text = re.sub(r'[0-9]',' ',text)
    text = text.translate(str.maketrans('', '', string.punctuation)).lower()
    tokens = word_tokenize(text)
    output = [word for word in tokens if word not in stop_words]
    return " ".join(output)
```

[55]:
```
preprocessed_books = []
for i in book_list:
  preprocessed_books.append(q4_preprocess(i))
```

[56]:
```
q4_vectorizer = CountVectorizer()

q4_dtm = q4_vectorizer.fit_transform(preprocessed_books)
vocab = q4_vectorizer.get_feature_names_out()
doc_labels = range(len(preprocessed_books))
q4_df = pd.DataFrame(q4_dtm.toarray(), columns=vocab, index=doc_labels)
q4_df
```

[56]:
```
   aaron  ab  aback  abandon  abandoned  abandoning  abash  abbey  abbot  \
0      0   0      0        1          0           0      0      0      0
1      0   1      0        0          1           0      1      2      1
2      0   0      0        0          1           1      0      0      0
3      1   1      1        0          1           0      0      0      0

   abbots  …  zed  zgerald  zis  zoe  zola  zoologi  zopy  zopyr  zopyrion  \
0       0  …    0        0    0    0     0        0     2      2        19
1       1  …    0        0    0    0     0        1     0      0         0
2       0  …    0        0    0    6     0        0     0      0         0
3       0  …    1        1    1    0     1        0     0      0         0

   zulu
0     0
1     0
2     1
3     0

[4 rows x 17253 columns]
```

[57]:
```
columns_to_keep = [col for col in q4_df.columns if (q4_df[col] != 0).sum() < 2]

q4_df.drop(columns=columns_to_keep, inplace=True)
q4_df
```

[57]:
```
   ab  abandoned  abide  ability  able  abo  abode  abou  abroad  abrupt  …  \
0   0          0      4        1     5    1      5     0       2       0  …
```

```
1   1           1       2         2      4     1        0      1         4         0  …
2   0           1       4         0      4     1        0      1         2         1  …
3   1           1       1         0     13     0        1      3         4         3  …

   york  youd  young  younger  youre  youth  youthful  youve  ything  zeal
0     0     0      7        2      0     11         1      0       0     2
1     3     1     25        1      1      0         1      1       1     2
2     1     0     43        1      0      3         0      0       0     1
3     0     1     52        0      1      3         1      3       1     0

[4 rows x 5644 columns]
```

##4 C

[58]:
```python
ratios = pd.DataFrame(index=q4_df.index, columns=q4_df.columns)

for index, row in q4_df.iterrows():
    sum_others = q4_df.drop(index).sum()

    avg_others = sum_others / 3

    ratios.loc[index] = row / avg_others

print(ratios)
```

```
    ab abandoned     abide ability      able  abo abode  abou abroad abrupt  \
0  0.0       0.0  1.714286     1.5  0.714286  1.5  15.0   0.0    0.6    0.0
1  3.0       1.5  0.666667     6.0  0.545455  1.5   0.0  0.75    1.5    0.0
2  0.0       1.5  1.714286     0.0  0.545455  1.5   0.0  0.75    0.6    1.0
3  3.0       1.5       0.3     0.0       3.0  0.0   0.6   4.5    1.5    9.0

   … york youd     young younger youre     youth youthful youve ything  zeal
0  …  0.0  0.0     0.175     3.0   0.0       5.5      1.5   0.0    0.0   2.0
1  …  9.0  3.0  0.735294     1.0   3.0       0.0      1.5   1.0    3.0   2.0
2  …  1.0  0.0  1.535714     1.0   0.0  0.642857      0.0   0.0    0.0  0.75
3  …  0.0  3.0      2.08     0.0   3.0  0.642857      1.5   9.0    3.0   0.0

[4 rows x 5644 columns]
```

[59]:
```python
author = ["Shelley, Mary Wollstonecraft","Twain, Mark","Joyce, James","Hume,␣
↪Fergus"]
q4_highest_word = []
for i in range(4):
  print(f"the top 5 words for {author[i]} are: \n{ratios.iloc[i].
↪sort_values(ascending=False).head(5)}\n")
  q4_highest_word.append(ratios.iloc[i].sort_values(ascending=False).index.
↪to_list())
```

```
the top 5 words for Shelley, Mary Wollstonecraft are:
```

```
fields      54.0
grief       45.0
wept        42.0
greater     42.0
rome        36.0
Name: 0, dtype: object

the top 5 words for Twain, Mark are:
tom           109.5
warn          108.0
jim            99.0
duke           66.0
presently      57.0
Name: 1, dtype: object

the top 5 words for Joyce, James are:
stephen     60.0
bloom       33.0
ject        33.0
shameful    27.0
michael     24.0
Name: 2, dtype: object

the top 5 words for Hume, Fergus are:
lucy       159.0
brian      117.0
madame      72.0
hale        54.0
someone     51.0
Name: 3, dtype: object
```

##4 D

[141]:
```
with open("/content/mystery-excerpt.txt","r") as file:
  q4_mystery = file.read()
```

[61]:
```
q4_preprocessed_mystery  = [q4_preprocess(q4_mystery)]
q4_dtm_mystery = q4_vectorizer.transform(q4_preprocessed_mystery)
vocab_mystery = q4_vectorizer.get_feature_names_out()
doc_labels_mystery = range(len(q4_preprocessed_mystery))
q4_mystery_df = pd.DataFrame(q4_dtm_mystery.toarray(), columns=vocab_mystery,
 ↪index=doc_labels_mystery)
q4_mystery_df
```

[61]:
```
   aaron  ab  aback  abandon  abandoned  abandoning  abash  abbey  abbot  \
0      0   0      0        0          0           0      0      0      0
```

```
   abbots  …  zed  zgerald  zis  zoe  zola  zoologi  zopy  zopyr  zopyrion  \
0       0  …    0        0    0    0     0        0     0      0         0

   zulu
0     0

[1 rows x 17253 columns]
```

[62]:
```python
q4_df.drop(2,axis='index')
```

[62]:
```
   ab  abandoned  abide  ability  able  abo  abode  abou  abroad  abrupt  …  \
0   0          0      4        1     5    1      5     0       2       0  …
1   1          1      2        2     4    1      0     1       4       0  …
3   1          1      1        0    13    0      1     3       4       3  …

   york  youd  young  younger  youre  youth  youthful  youve  ything  zeal
0     0     0      7        2      0     11         1      0       0     2
1     3     1     25        1      1      0         1      1       1     2
3     0     1     52        0      1      3         1      3       1     0

[3 rows x 5644 columns]
```

[63]:
```python
result = pd.DataFrame(columns=["Author", "Word", "Chi2", "P value"])
for k,j in enumerate(q4_highest_word):
  for i in range(len(j)):
    auth_word_pair = {}
    tp_characteristic_tokens = sum(q4_mystery_df[j[i]])
    tn_characteristic_tokens = q4_mystery_df.to_numpy().sum() -␣
 ↪tp_characteristic_tokens

    fp_characteristic_tokens = sum(q4_df.drop(k,axis='index')[j[i]])
    fn_characteristic_tokens = q4_df.drop(k,axis='index').to_numpy().sum() -␣
 ↪fp_characteristic_tokens

    observed = [[tp_characteristic_tokens, tn_characteristic_tokens],
                [fp_characteristic_tokens, fn_characteristic_tokens]]

    # Perform chi-squared test
    chi2, p, _, _ = chi2_contingency(observed)
    result = pd.concat([result,pd.DataFrame({"Author": [author[k]], "Word":␣
 ↪[j[i]], "Chi2":[chi2], "P value": [p]})], ignore_index = True)
```

[64]:
```python
result_filter = result[result["P value"] != "0.0"]
best = result_filter["Chi2"].argmax()
best_row = result_filter.iloc[best]
best_row
```

[64]:
```
Author     Joyce, James
Word        mississippi
Chi2         502.582289
P value             0.0
Name: 12433, dtype: object
```

Since selecting only the top 5 words didn't yield any results because those words weren't present in the mystery text, weconsidered all the words. We'll identify the word with the highest chi-square value and p-value.

In this case, because the data lines are randomly selected, we can't consistently determine the correct author every time. Although the prediction for the mystery text suggests it's by Joyce, James, and the word highly associated with it is "Mississippi," this may not always be accurate. This is because the 500 random lines we gather from each book might not be sufficient to make a perfect prediction.

#Question 5

##5A

[69]:
```
import os
import string

un = []

for z in os.listdir("/content/UN"):
  with open(os.path.join("/content/UN",z), 'r') as content_file:
    content = content_file.read()
  un.append(content)
```

[70]:
```
#preprocessing
def q5_preprocess(doc):
  d = doc.translate(str.maketrans('', '', string.punctuation)).lower()
  tokens = word_tokenize(d)
  output = ' '.join(tokens)
  return output

q5_preprocessed_corpus = list(map(q5_preprocess, un))
```

[71]:
```
#creating dtm
vector_q5 = CountVectorizer()
dtm_q5 = vector_q5.fit_transform(q5_preprocessed_corpus)
dtm_q5.toarray()
```

[71]:
```
array([[0, 0, 0, …, 0, 0, 0],
       [0, 0, 0, …, 0, 0, 0],
       [0, 0, 0, …, 0, 0, 0],
       …,
       [0, 1, 0, …, 0, 0, 0],
```

```
        [0, 0, 0, …, 0, 0, 0],
        [0, 0, 0, …, 0, 0, 0]])
```

[72]:
```
new=" ".join(q5_preprocessed_corpus)
new
word_tokens = nltk.word_tokenize(new)
q5_finder = BigramCollocationFinder.from_words(word_tokens)

min_freq = 5
q5_finder.apply_freq_filter(min_freq)
```

[73]:
```
bigram_freq = q5_finder.ngram_fd

united_nations_freq = bigram_freq[('united', 'nations')]

print("The frequency of united nations under independence is {}".
 ↪format(united_nations_freq))
```

```
The frequency of united nations under independence is 1916
```

Calculating contingency table for the bigram "United Nations"

[74]:
```
bigrams_starting_with_united = 0

for bigram, freq in q5_finder.ngram_fd.items():
    # Check if the first word of the bigram is 'united' and the second is not
 ↪'nations'
    if bigram[0].lower() == 'united' and bigram[1].lower() != 'nations':
        bigrams_starting_with_united += freq

print("The number of bigrams starting with united is {}".
 ↪format(bigrams_starting_with_united))
```

```
The number of bigrams starting with united is 324
```

[75]:
```
bigrams_ending_with_nations = 0

for bigram, freq in q5_finder.ngram_fd.items():
    # Check if the first word of the bigram is 'united' and the second is not
 ↪'nations'
    if bigram[0].lower() != 'united' and bigram[1].lower() == 'nations':
        bigrams_ending_with_nations += freq

print("The number of bigrams ending with nations is {}".
 ↪format(bigrams_ending_with_nations))
```

```
The number of bigrams ending with nations is 236
```

[76]:
```
non_united_nations_bigrams = 0

for bigram, freq in q5_finder.ngram_fd.items():
    # Check if the first word of the bigram is 'united' and the second is not␣
↪'nations'
    if bigram[0].lower() != 'united' and bigram[1].lower() != 'nations':
        non_united_nations_bigrams += freq

print("The number of bigrams not containing united or nations is {}".
↪format(non_united_nations_bigrams))
```

```
The number of bigrams not containing united or nations is 237369
```

[77]:
```
#constructing contingency table
observed = [[united_nations_freq, bigrams_starting_with_united],

            [bigrams_ending_with_nations, non_united_nations_bigrams]]
```

[78]:
```
print("The contingency table is {}". format(observed))
```

```
The contingency table is [[1916, 324], [236, 237369]]
```

[80]:
```
# getting the expected frequency
united_prob = word_tokens.count("united") / len(word_tokens)
nations_prob = word_tokens.count("nations") / len(word_tokens)
expected_freq = united_prob * nations_prob * len(word_tokens)
print(f"The expected frequency of the bigram united nations with independance␣
↪is : {expected_freq}")
print(f"The observed frequency of the bigram united nations is :␣
↪{united_nations_freq}")
```

```
The expected frequency of the bigram united nations with independance is :
13.08967392662724
The observed frequency of the bigram united nations is : 1916
```

Since the Observed Frequency of united nations is far greater than the Expected frequency, it is a bigram.

##5 B

[37]:
```
scored = q5_finder.score_ngrams(BigramAssocMeasures.likelihood_ratio)

top_10_collocations = scored[:10]

# Print the top 10 collocations with their scores
for collocation, score in top_10_collocations:
    print(collocation, score)
```

```
('united', 'nations') 20226.044907531486
('of', 'the') 8784.957184981362
```

```
('the', 'united') 7845.187167522721
('climate', 'change') 7286.292867416467
('sustainable', 'development') 6036.140881743866
('it', 'is') 5706.791078038168
('human', 'rights') 5100.379045776574
('general', 'assembly') 5007.354918454996
('we', 'are') 4514.764574240972
('security', 'council') 4495.41714551169
```

[38]:
```
raw_freq_scores = q5_finder.score_ngrams(BigramAssocMeasures.raw_freq)

print("Raw Frequency Scores:", raw_freq_scores[:10])
```

```
Raw Frequency Scores: [(('of', 'the'), 0.013209395791988883), (('in', 'the'),
0.0056436737366718824), (('united', 'nations'), 0.0047888746144656), (('the',
'united'), 0.004738886361705), (('to', 'the'), 0.004461451558883662), (('and',
'the'), 0.004081540837903093), (('on', 'the'), 0.003054282243672737), (('for',
'the'), 0.002621883857293536), (('it', 'is'), 0.0024844161622018827), (('we',
'are'), 0.00237694141876659)]
```

Most of the common bigrams are with stop words. Other than stop words, we see Uited Nations here again meaning that it is a plaussible bigram.

From the top ten collocations analyzed, we can deduce that: - "United Nations" plays a pivotal role as a multi-word term, as previously mentioned. - "of the" could be seen as either significant or not, being merely a conjunction of two common stop words without forming a meaningful multi-word expression in practical scenarios. - "the united" may not hold significance on its own because it presumably precedes "nations" to form "the United Nations." Given that "United Nations" stands as a multi-word entity, the inclusion of "the united" as a separate multi-word doesn't seem justified. Therefore, "the United Nations" is better recognized as a trigram multi-word. - "Climate change" is identified as a critical multi-word due to its high relevance score and the coherent combination of both terms into a bigram. - Similarly, "sustainable development" is acknowledged as a crucial multi-word for analogous reasons mentioned above. - "it is" and "we are," akin to "of the," may or may not be deemed essential, following the same logic applied to the bigram "of the." - "Human rights," "general assembly," and "security council" are all considered essential bigrams within the context of our dataset.

[82]:
```
raw_freq = q5_finder.score_ngrams(BigramAssocMeasures.raw_freq)
print("Raw Frequency Scores:", raw_freq[:10])
```


```
Raw Frequency Scores: [(('of', 'the'), 0.013209395791988883), (('in', 'the'),
0.0056436737366718824), (('united', 'nations'), 0.0047888746144656), (('the',
'united'), 0.004738886361705), (('to', 'the'), 0.004461451558883662), (('and',
'the'), 0.004081540837903093), (('on', 'the'), 0.003054282243672737), (('for',
'the'), 0.002621883857293536), (('it', 'is'), 0.0024844161622018827), (('we',
'are'), 0.00237694141876659)]
```


#Question 6

##6 A

[86]:
```
# List of book IDs
book_ids = [[4217], [74]]

collected_texts = []

# Loop through each ID list
for single_list in book_ids:
    author_text = ""
    for book_id in single_list:
        url = f"https://www.gutenberg.org/cache/epub/{book_id}/pg{book_id}.txt"
        response = requests.get(url)
        text_snippet = ""
        text_snippet += str(list(response)[100:])
        author_text += text_snippet
    collected_texts.append(author_text)  # Adding the aggregated text to the␣
↪main list
```

[87]:
```
len(collected_texts[1])
```

[87]: 512403

I need to implement Zipf's Law, which posits that the frequency of occurrence of the (i^{th}) most common word in a corpus is inversely proportional to its rank ((i)). To achieve this, I plan to sum the values column-wise and store the results in a DataFrame. This process will involve manipulating data to reflect the principle that each term's frequency is a function of its inverse rank. After completing the calculations and analysis, I intend to remove this segment of code or data for clarity or confidentiality.

[88]:
```
def q6_preprocess(text):
    escapes = {
        '\\a': ' ', '\\b': ' ', '\\f': ' ', '\\n': ' ',
        '\\r': ' ', '\\t': ' ', '\\v': ' ', "\\'": ' ',
        '\\"': ' ', '\\\\': ' ', '\\?': " ", "b\'": " "
    }
    for escape, replacement in escapes.items():
        text = text.replace(escape, replacement)
    text = re.sub(r'\\x[0-9A-Fa-f]{2}', ' ', text)
    text = re.sub(r'[0-9]', ' ', text)
    text = text.translate(str.maketrans('', '', string.punctuation)).lower()
    tokens = word_tokenize(text)
    filtered_tokens = [word for word in tokens if word not in stop_words]
    stemmed_tokens = [PorterStemmer().stem(word) for word in filtered_tokens]
    return " ".join(stemmed_tokens)
```

[90]:
```
cleaned_texts = []
for text in collected_texts:
  cleaned_texts.append(q6_preprocess(text))
```

[92]:
```
q6_text_vectorizer = CountVectorizer()

# Creating the document-term matrix (DTM)
q6_dtm = q6_text_vectorizer.fit_transform(cleaned_texts)
q6_vocabulary = q6_text_vectorizer.get_feature_names_out()

# Creating document labels based on the number of preprocessed texts
q6_document_labels = range(len(cleaned_texts))

# Constructing the DataFrame from the DTM
q6_dtm_df = pd.DataFrame(q6_dtm.toarray(), columns=q6_vocabulary,␣
↪index=q6_document_labels)
q6_dtm_df
```

[92]:
```
   aband  abandon  abas  abash  abbey  abbot  abet  abhor  abid  abit  …  \
0      1        4     4      0      1      1     1      1     2     0  …
1      0        4     0      2      0      0     0      0     2     1  …

   ys  zacchari  zeal  zealou  zebra  zed  zenith  zephyr  zincroof  zoolog
0   0         1     2       2      0    1       0       0         1       1
1   3         0     1       0      1    0       1       1         0       0

[2 rows x 10646 columns]
```

[93]:
```
q6_column_sums = q6_dtm_df.sum()

# Create a new DataFrame with column names and their corresponding sums
q6_result_df = pd.DataFrame({
    'Term': q6_column_sums.index,
    'Frequency': q6_column_sums.values
})

# Show the resulting DataFrame
print(q6_result_df)
```

```
            Term  Frequency
0          aband          1
1        abandon          8
2           abas          4
3          abash          2
4          abbey          1
…            …        …
10641        zed          1
10642     zenith          1
10643     zephyr          1
10644   zincroof          1
10645     zoolog          1
```

```
[10646 rows x 2 columns]
```

[94]:
```python
q6_sorted_result_df = q6_result_df.sort_values(by='Frequency', ascending=False)

# Print the sorted DataFrame
print(q6_sorted_result_df)
```

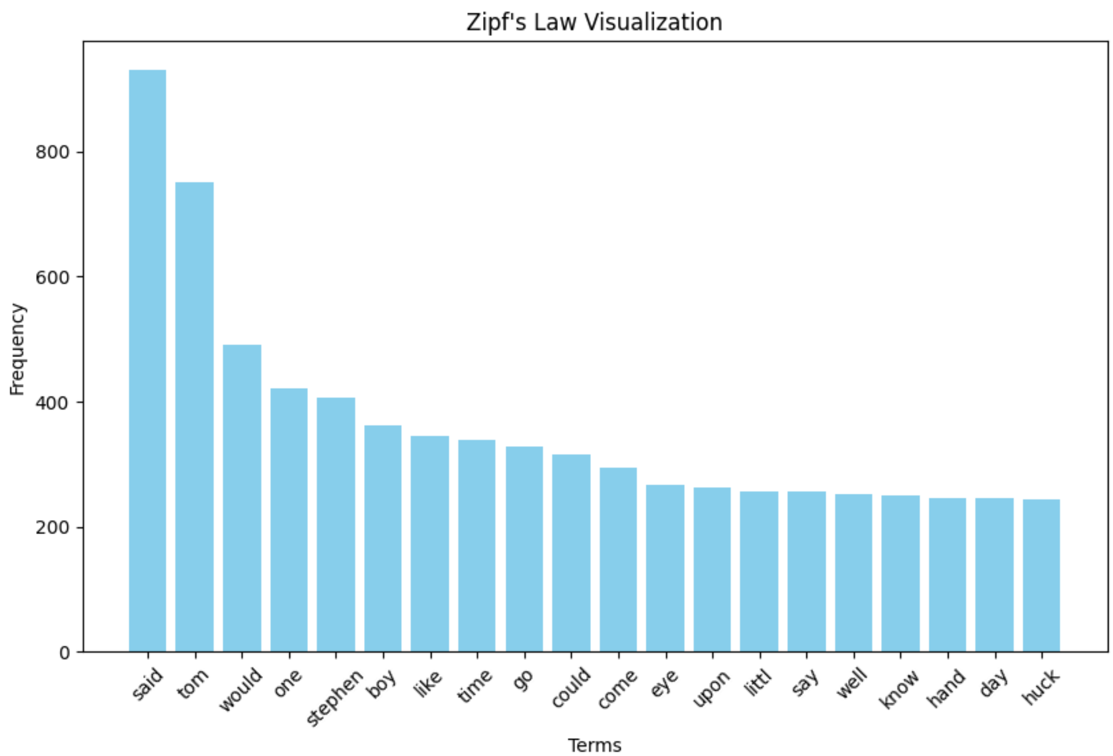
```
             Term  Frequency
7886         said        930
9475          tom        751
10531       would        491
6387          one        422
8807      stephen        406
…             …          …
4916        ivoir          1
4910   ittingroom          1
4909        itter          1
4908          itl          1
10645      zoolog          1

[10646 rows x 2 columns]
```

[95]:
```python
q6_sorted_result_index_df = q6_sorted_result_df.reset_index(drop=True)
q6_zipfs_df = q6_sorted_result_df[0:20]
q6_zipfs_df_reset_index = q6_zipfs_df.reset_index(drop=True)
q6_zipfs_df_reset_index
```

[95]:

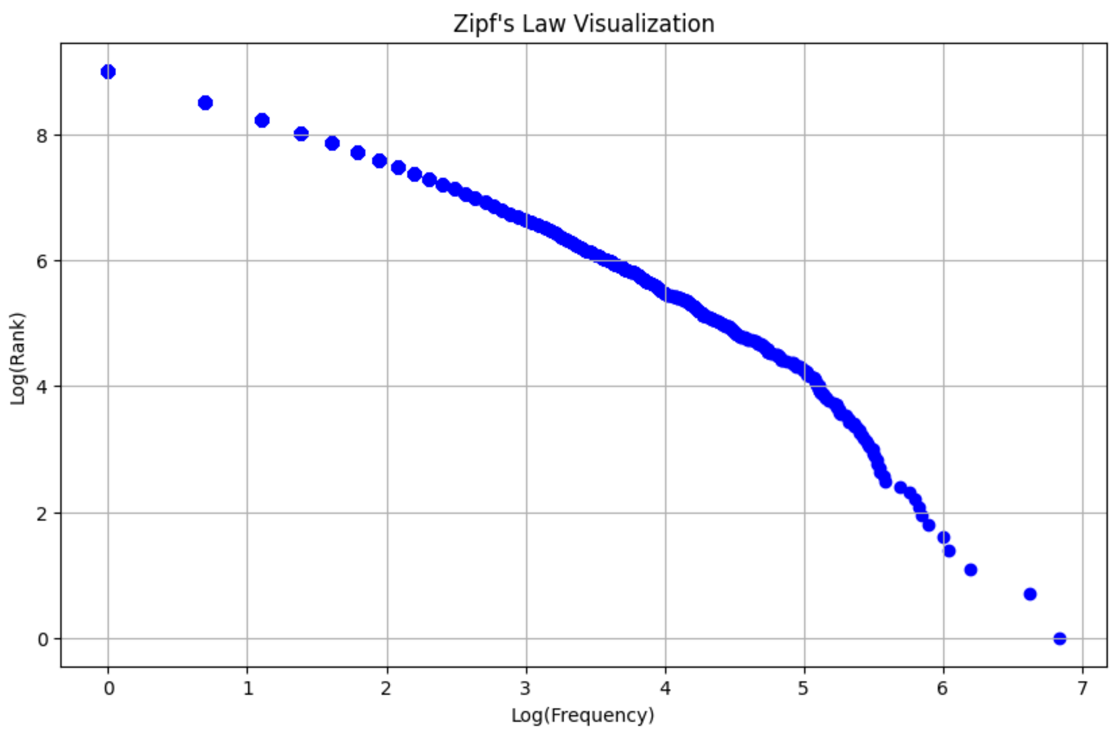
	Term	Frequency
0	said	930
1	tom	751
2	would	491
3	one	422
4	stephen	406
5	boy	362
6	like	346
7	time	339
8	go	329
9	could	316
10	come	295
11	eye	266
12	upon	263
13	littl	257
14	say	256
15	well	252
16	know	250
17	hand	246
18	day	245
19	huck	244

[96]:
```
plt.figure(figsize=(10, 6))
plt.bar(q6_zipfs_df_reset_index['Term'], q6_zipfs_df_reset_index['Frequency'],␣
 ↪color='skyblue')
plt.xlabel('Terms')
plt.ylabel('Frequency')
plt.title('Zipf\'s Law Visualization')
plt.xticks(rotation=45)
plt.show()
```



[97]:
```
q6_sorted_result_index_df['Rank'] = q6_sorted_result_index_df['Frequency'].
 ↪rank(ascending=False)

# Applying logarithm transformation to frequency and rank
q6_sorted_result_index_df['log(Frequency)'] = np.
 ↪log(q6_sorted_result_index_df['Frequency'])
q6_sorted_result_index_df['log(Rank)'] = np.
 ↪log(q6_sorted_result_index_df['Rank'])

# Plotting the Zipf's Law graph
plt.figure(figsize=(10, 6))
plt.scatter(q6_sorted_result_index_df['log(Frequency)'],␣
 ↪q6_sorted_result_index_df['log(Rank)'], color='blue')
```

```
plt.title("Zipf's Law Visualization")
plt.xlabel('Log(Frequency)')
plt.ylabel('Log(Rank)')
plt.grid(True)
plt.show()
```



##6 B

To calculate the 'b' value, we'll utilize Heap's Law, which is formulated as M = k * (T^b), where:

- M represents the number of types,
- T is the number of tokens,
- k (=44) and b are constants that vary with the language.

By logarithmically transforming both sides, we get log(M) = log(k) + b * log(T). This allows us to isolate b and compute it as b = log(M / k) / log(T).

[98]:
```
stop_words = set(stopwords.words("english"))# removing stop words to get actual␣
↪words used
def preprocess_6b(text):
    escapes = {
          '\\a': ' ', '\\b': ' ', '\\f': ' ', '\\n': ' ',
          '\\r': ' ', '\\t': ' ', '\\v': ' ', "\\'": ' ',
          '\\"': ' ', '\\\\': ' ', '\\?':" " ,"b\'":" "
      }
```

```
    for escape, replacement in escapes.items():
        text = text.replace(escape, replacement)
    text = re.sub(r'\\x[0-9A-Fa-f]{2}', ' ', text)
    text = re.sub(r'[0-9]',' ',text)
    text = text.translate(str.maketrans('', '', string.punctuation)).lower()
    tokens = word_tokenize(text)
    output = [word for word in tokens if word not in stop_words]
    return " ".join(output)
```

[105]:
```
q6b_tokens=word_tokenize(cleaned_texts[0]+cleaned_texts[1])
q6b_total_types=len(set(q6b_tokens))
q6b_total_tokens=len(q6b_tokens)
b=(np.log(q6b_total_types)-np.log(44))/(np.log(q6b_total_tokens))
print("The value of b that best first both the novels is {}". format(b) )
```

```
The value of b that best first both the novels is 0.4860252025361327
```

#Question 7

##7 A

First, I have unzipped the folder with all the inaugral speeches between 1913 and 2021. Then, I basically tried to divide the speeches of each president into chunks of 150 but the last one. The last chunk would have 150+remaining tokens if they are less than 150 (to form a new chunk). I displayed those chunk results in the following code cells.

[9]:
```
# Function to tokenize and chunk the text
def tokenize_and_chunk(text, chunk_size=150):
    chunked_list=[]
    tokens = word_tokenize(text)
    len_tokens=len(tokens)
    return [tokens[i:i + chunk_size] for i in range(0, len(tokens), chunk_size)]

tokenized_speeches = {}
data_dir_path = Path('/content/q7_data')

for file_path in data_dir_path.glob('*.txt'):
    with open(file_path, 'r') as file:
        speech_content = file.read()
        chunks = tokenize_and_chunk(speech_content)
        if len(chunks[-1])<150:
          chunks[-2]=chunks[-1]+chunks[-2]
          chunks=chunks[:-1]
        tokenized_speeches[file_path.stem] = chunks

print(f"Number of speeches processed: {len(tokenized_speeches)}")
```

```
Number of speeches processed: 28
```

[10]:
```
len(chunks[-1])
```

[10]: 176

[11]:
```
len(chunks[-2])
```

[11]: 150

[12]:
```
sorted(tokenized_speeches.keys())
```

[12]:
```
['1913-Wilson',
 '1917-Wilson',
 '1921-Harding',
 '1925-Coolidge',
 '1929-Hoover',
 '1933-Roosevelt',
 '1937-Roosevelt',
 '1941-Roosevelt',
 '1945-Roosevelt',
 '1949-Truman',
 '1953-Eisenhower',
 '1957-Eisenhower',
 '1961-Kennedy',
 '1965-Johnson',
 '1969-Nixon',
 '1973-Nixon',
 '1977-Carter',
 '1981-Reagan',
 '1985-Reagan',
 '1989-Bush',
 '1993-Clinton',
 '1997-Clinton',
 '2001-Bush',
 '2005-Bush',
 '2009-Obama',
 '2013-Obama',
 '2017-Trump',
 '2021-Biden']
```

[13]:
```
!pip install textstat
```

```
Requirement already satisfied: textstat in /usr/local/lib/python3.10/dist-
packages (0.7.3)
Requirement already satisfied: pyphen in /usr/local/lib/python3.10/dist-packages
(from textstat) (0.14.0)
```

##7 B

Calculate an estimated syllable count for a given word. This method employs a basic heuristic primarily based on vowel counting, with an adjustment for trailing 'e'.

Note: I have executed 7 B using readability instance as well. That would take longer time to execute.

This particular way of counting syllables is not the same as readability but when i compared the FRE scores later on, they are very approximately the same (+/- 10 points). So I decided to keep this code as this executes 60x faster.

[14]:
```
def count_syllables(word):
    word = word.lower()
    vowels = "aeiou"
    count = sum(letter in vowels for letter in word)  # Count vowels in the word

    # Deduct one for silent ending 'e'
    if word.endswith('e'):
        count -= 1
    return max(1, count)
```

Calculate the Flesch Reading Ease score for a given text.

[15]:
```
def calculate_flesch_score(text):
    text_str = ' '.join(text)
    words = re.findall(r'\w+', text_str)
    sentences = re.split(r'[.!?]+', text_str)[:-1]  # Exclude the last empty␣
 ↪split
    num_words = len(words)
    num_sentences = len(sentences)
    num_syllables = sum(count_syllables(word) for word in words)

    # Flesch Reading Ease formula
    flesch_score = 206.835 - 1.015 * (num_words / max(1, num_sentences)) - 84.6␣
 ↪* (num_syllables / num_words)
    return flesch_score
```

Estimate the mean Flesch Reading Ease score for given text chunks using bootstrap sampling.

[16]:
```
def calculate_mean_fre(scores):
    return np.mean(scores)

def bootstrap_fre(text_chunks, iterations=1000):
    """
    Perform bootstrapping to estimate the mean Flesch Reading Ease score
    over a specified number of iterations.
    """
    bootstrapped_means = []
    for _ in range(iterations):
        sampled_fre_scores = [calculate_flesch_score(random.
 ↪choice(text_chunks)) for _ in text_chunks]
        bootstrapped_means.append(calculate_mean_fre(sampled_fre_scores))
    return calculate_mean_fre(bootstrapped_means)

bootstrapped_fre_scores = {}
```

```
for speech, chunks in tokenized_speeches.items():
    mean_fre = bootstrap_fre(chunks)
    bootstrapped_fre_scores[speech] = mean_fre

print("Bootstrapped FRE scores (mean):")
for speech, score in bootstrapped_fre_scores.items():
    print(f"{speech}: {score}")
```

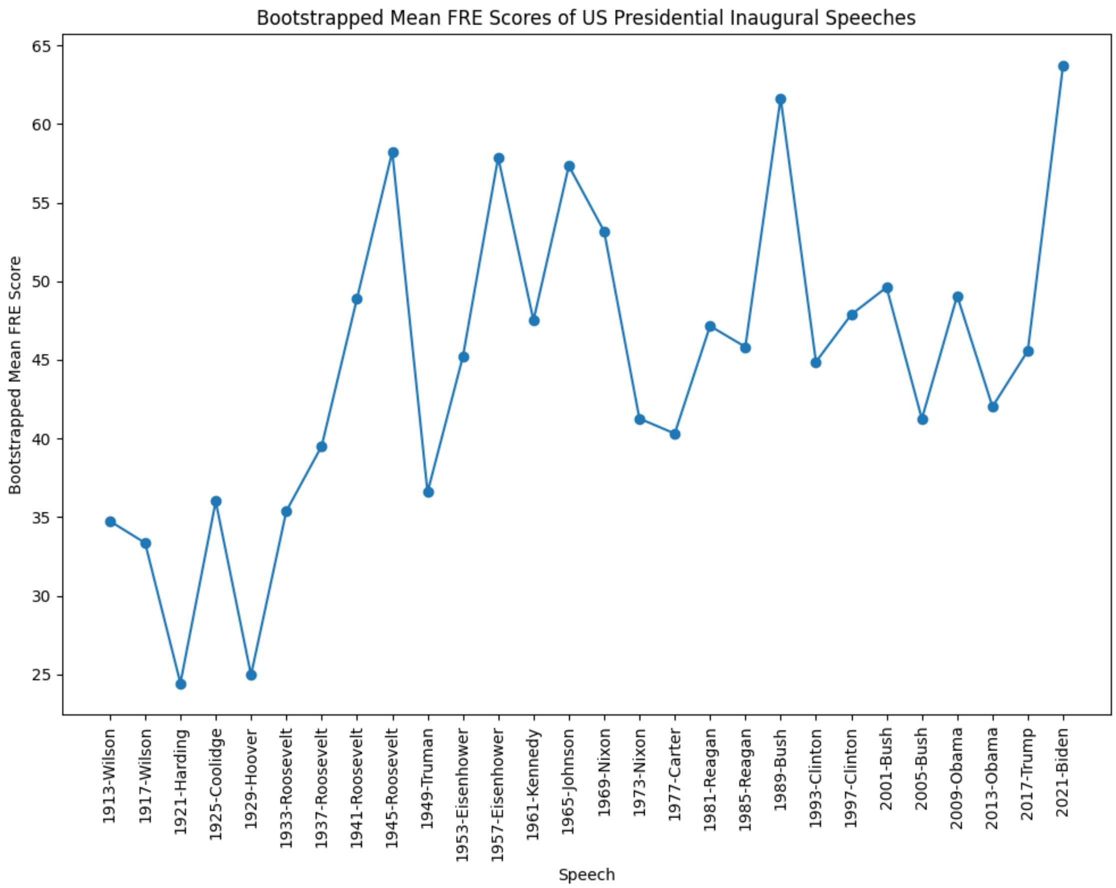
```
Bootstrapped FRE scores (mean):
2009-Obama: 49.0781075555217
1977-Carter: 40.31201425350032
1949-Truman: 36.57973225083378
1989-Bush: 61.65175799052369
1973-Nixon: 41.26889845187607
1917-Wilson: 33.35306723795411
1969-Nixon: 53.18353315520894
1961-Kennedy: 47.51286751813469
2001-Bush: 49.61511514662368
1929-Hoover: 24.953635695199605
1933-Roosevelt: 35.381477341245784
2017-Trump: 45.57554012879571
1953-Eisenhower: 45.20226265974052
1925-Coolidge: 36.002490058650224
1985-Reagan: 45.82749079247133
2005-Bush: 41.247772085796456
1937-Roosevelt: 39.51117117888521
1945-Roosevelt: 58.23927012947307
2013-Obama: 42.02635740115898
1913-Wilson: 34.74778726538893
2021-Biden: 63.750745789605176
1921-Harding: 24.43533724417742
1993-Clinton: 44.873975541070834
1957-Eisenhower: 57.87572210207156
1981-Reagan: 47.15672278520609
1997-Clinton: 47.882719681868046
1965-Johnson: 57.34479993215627
1941-Roosevelt: 48.88950926052241
```

##7 C

Plotting the FRE scores vs President speeches scatter plot to show the trend of change in FRE.

[17]:
```
sorted_speeches = sorted(bootstrapped_fre_scores.items(), key=lambda x: x[0])
speeches, scores = zip(*sorted_speeches)

plt.figure(figsize=(10, 8))
plt.plot(speeches, scores, marker='o', linestyle='-')
plt.xticks(rotation=90)
```

```python
plt.xlabel('Speech')
plt.ylabel('Bootstrapped Mean FRE Score')
plt.title('Bootstrapped Mean FRE Scores of US Presidential Inaugural Speeches')
plt.tight_layout()
plt.show()
```



##7 D

Calculating the Mean FRE od the chunks of all the speeches.

[18]:
```python
def calculate_observed_flesch_scores(chunks):
    flesch_scores = [calculate_flesch_score(chunk) for chunk in chunks]
    return np.mean(flesch_scores)

observed_flesch_results = {}
for speech_title, chunks in tokenized_speeches.items():
    observed_mean = calculate_observed_flesch_scores(chunks)
    observed_flesch_results[speech_title] = observed_mean

print("Observed FRE scores (mean):")
```

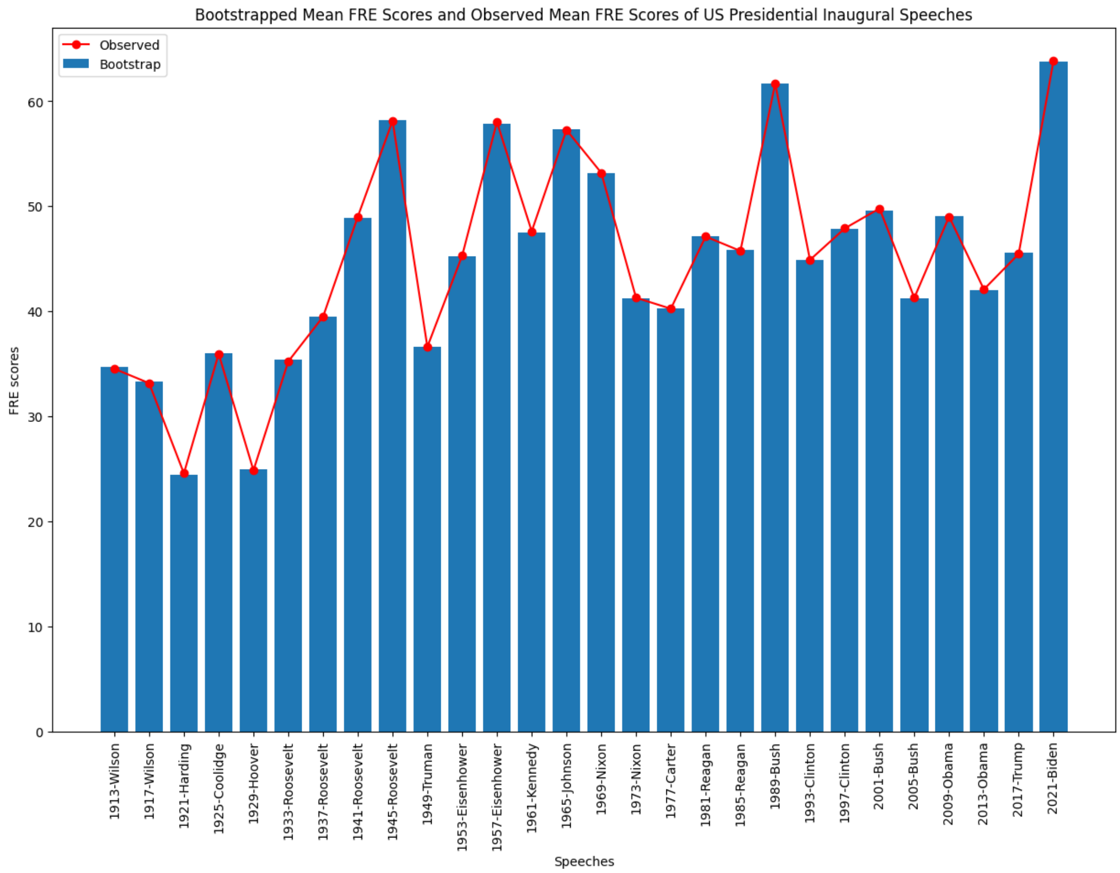
```
for title, mean_score in observed_flesch_results.items():
    print(f"{title}: {mean_score}")
```

```
Observed FRE scores (mean):
2009-Obama: 49.018219693968554
1977-Carter: 40.246455394027635
1949-Truman: 36.61473497676544
1989-Bush: 61.68711224006637
1973-Nixon: 41.29040091526095
1917-Wilson: 33.15942678702132
1969-Nixon: 53.18350016671256
1961-Kennedy: 47.61791748545477
2001-Bush: 49.792097741986936
1929-Hoover: 24.85810041152146
1933-Roosevelt: 35.22037461723786
2017-Trump: 45.46370586934774
1953-Eisenhower: 45.29438031430314
1925-Coolidge: 35.95045703542373
1985-Reagan: 45.74844643419749
2005-Bush: 41.28770265391407
1937-Roosevelt: 39.4818880469906
1945-Roosevelt: 58.136417804235265
2013-Obama: 42.08779061804608
1913-Wilson: 34.52695804152225
2021-Biden: 63.83118793263736
1921-Harding: 24.572886527009832
1993-Clinton: 44.89492081673787
1957-Eisenhower: 58.019753581489546
1981-Reagan: 47.11755801336833
1997-Clinton: 47.89382414844888
1965-Johnson: 57.27134244252781
1941-Roosevelt: 49.00459050056261
```

[19]:
```
observed_flesch_results = dict(sorted(observed_flesch_results.items()))
bootstrapped_fre_scores = dict(sorted(bootstrapped_fre_scores.items()))
```

[20]:
```
plt.figure(figsize=(15, 10))
plt.bar(bootstrapped_fre_scores.keys(),bootstrapped_fre_scores.
 ↪values(),label="Bootstrap")
plt.plot(observed_flesch_results.keys(),observed_flesch_results.
 ↪values(),color='r',marker="o",label="Observed")
plt.xticks(np.arange(len(bootstrapped_fre_scores.
 ↪keys())),bootstrapped_fre_scores.keys(), rotation=90)
plt.xlabel("Speeches")
plt.ylabel("FRE scores")
plt.title('Bootstrapped Mean FRE Scores and Observed Mean FRE Scores of US␣
 ↪Presidential Inaugural Speeches')
```

```
plt.legend(loc="upper left")
plt.show()
```



The Observed and Bootstrap FRE scores are approximately the same.

##7 E

[21]:
```
observed_flesch_results
```

[21]:
```
{'1913-Wilson': 34.52695804152225,
 '1917-Wilson': 33.15942678702132,
 '1921-Harding': 24.572886527009832,
 '1925-Coolidge': 35.95045703542373,
 '1929-Hoover': 24.85810041152146,
 '1933-Roosevelt': 35.22037461723786,
 '1937-Roosevelt': 39.4818880469906,
 '1941-Roosevelt': 49.00459050056261,
 '1945-Roosevelt': 58.136417804235265,
 '1949-Truman': 36.61473497676544,
 '1953-Eisenhower': 45.29438031430314,
```

```
'1957-Eisenhower': 58.019753581489546,
'1961-Kennedy': 47.61791748545477,
'1965-Johnson': 57.27134244252781,
'1969-Nixon': 53.18350016671256,
'1973-Nixon': 41.29040091526095,
'1977-Carter': 40.246455394027635,
'1981-Reagan': 47.11755801336833,
'1985-Reagan': 45.74844643419749,
'1989-Bush': 61.68711224006637,
'1993-Clinton': 44.89492081673787,
'1997-Clinton': 47.89382414844888,
'2001-Bush': 49.792097741986936,
'2005-Bush': 41.28770265391407,
'2009-Obama': 49.018219693968554,
'2013-Obama': 42.08779061804608,
'2017-Trump': 45.46370586934774,
'2021-Biden': 63.83118793263736}
```

[22]:
```
tokenized_speeches = dict(sorted(tokenized_speeches.items()))
```

[23]:
```
dale_chall = {}
for title, chunk in tokenized_speeches.items():
  readability_instances = [Readability(" ".join(text)) for text in chunk]
  dale_chall_readability_scores = [instance.dale_chall().score for instance in␣
↪readability_instances]
  dale_chall[title] = np.mean(dale_chall_readability_scores)
for title, mean_score in dale_chall.items():
    print(f"{title}: {mean_score}")
```

```
1913-Wilson: 7.975710960089025
1917-Wilson: 7.944532564599703
1921-Harding: 9.113180467669576
1925-Coolidge: 8.467908965839527
1929-Hoover: 9.0033881926002
1933-Roosevelt: 8.628369705744799
1937-Roosevelt: 8.332942390689263
1941-Roosevelt: 7.470805068914652
1945-Roosevelt: 7.15995488321107
1949-Truman: 8.564997385546357
1953-Eisenhower: 8.085282262346846
1957-Eisenhower: 7.312416364349623
1961-Kennedy: 7.964062643070484
1965-Johnson: 6.755257933712714
1969-Nixon: 6.930360184388553
1973-Nixon: 7.705058154777471
1977-Carter: 8.22260203519538
1981-Reagan: 7.715048355013993
1985-Reagan: 7.506660840914468
```

```
1989-Bush: 6.87276255659249
1993-Clinton: 7.356359729819794
1997-Clinton: 7.396787447182929
2001-Bush: 7.226084979325138
2005-Bush: 7.633545084996106
2009-Obama: 7.43605839399154
2013-Obama: 7.763494414077697
2017-Trump: 6.823126141368596
2021-Biden: 7.005055983909582
```

[24]:
```python
dale_chall = dict(sorted(dale_chall.items()))
observed_fre_scores = dict(sorted(observed_flesch_results.items()))
readability_scores_df = pd.DataFrame({
    "Speech_Name": observed_fre_scores.keys(),
    "Flesch_Score": observed_fre_scores.values(),
    "Dale_Chall_Score": dale_chall.values()
})
readability_scores_df
```

[24]:

	Speech_Name	Flesch_Score	Dale_Chall_Score
0	1913-Wilson	34.526958	7.975711
1	1917-Wilson	33.159427	7.944533
2	1921-Harding	24.572887	9.113180
3	1925-Coolidge	35.950457	8.467909
4	1929-Hoover	24.858100	9.003388
5	1933-Roosevelt	35.220375	8.628370
6	1937-Roosevelt	39.481888	8.332942
7	1941-Roosevelt	49.004591	7.470805
8	1945-Roosevelt	58.136418	7.159955
9	1949-Truman	36.614735	8.564997
10	1953-Eisenhower	45.294380	8.085282
11	1957-Eisenhower	58.019754	7.312416
12	1961-Kennedy	47.617917	7.964063
13	1965-Johnson	57.271342	6.755258
14	1969-Nixon	53.183500	6.930360
15	1973-Nixon	41.290401	7.705058
16	1977-Carter	40.246455	8.222602
17	1981-Reagan	47.117558	7.715048
18	1985-Reagan	45.748446	7.506661
19	1989-Bush	61.687112	6.872763
20	1993-Clinton	44.894921	7.356360
21	1997-Clinton	47.893824	7.396787
22	2001-Bush	49.792098	7.226085
23	2005-Bush	41.287703	7.633545
24	2009-Obama	49.018220	7.436058
25	2013-Obama	42.087791	7.763494
26	2017-Trump	45.463706	6.823126

```
27       2021-Biden     63.831188          7.005056
```

[25]:
```
flesch_dale_chall_correlation = readability_scores_df['Flesch_Score'].
 ↪corr(readability_scores_df['Dale_Chall_Score'])
print("Correlation between FRE and Dale-Chall is :",␣
 ↪flesch_dale_chall_correlation)
```

```
Correlation between FRE and Dale-Chall is : -0.8694468462081306
```