Assignment 6

Thijs van der Knaap (s2752077)
Hatim Alsayahani (s3183696)
Sebastian Wehkamp (s2589907)
Dimitris Laskaratos (s3463702)

Group 15

October 16, 2017

Contents

1 State of Corpus

1.1 Term frequency count

In figure 1 and 2 you see the frequency of the 50 most frequent words without removing stop-words. The frequency is calculated with *nltk.FreqDist*

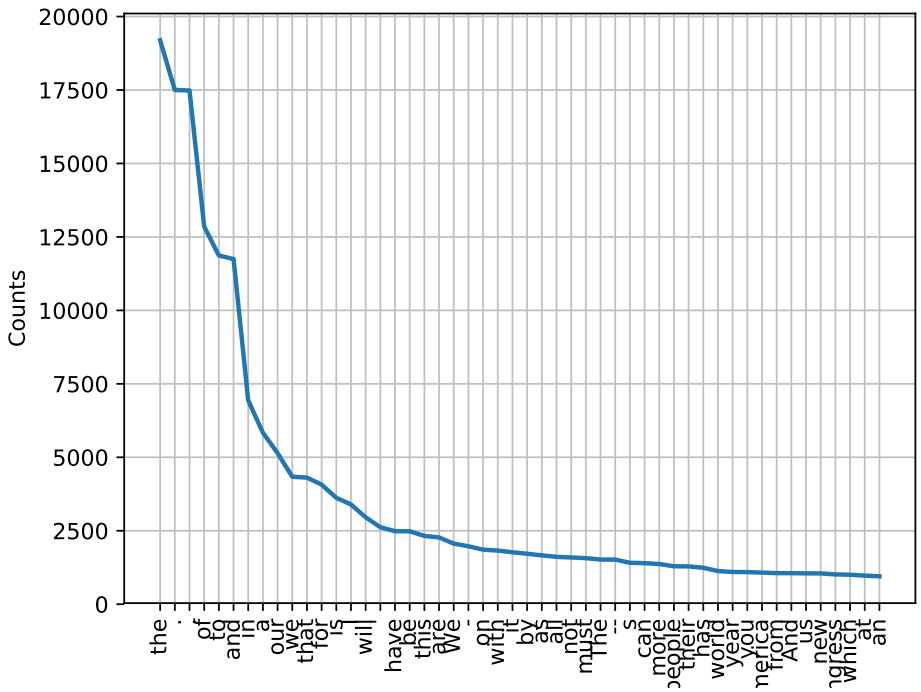

Figure 1: Frequency plot state_union

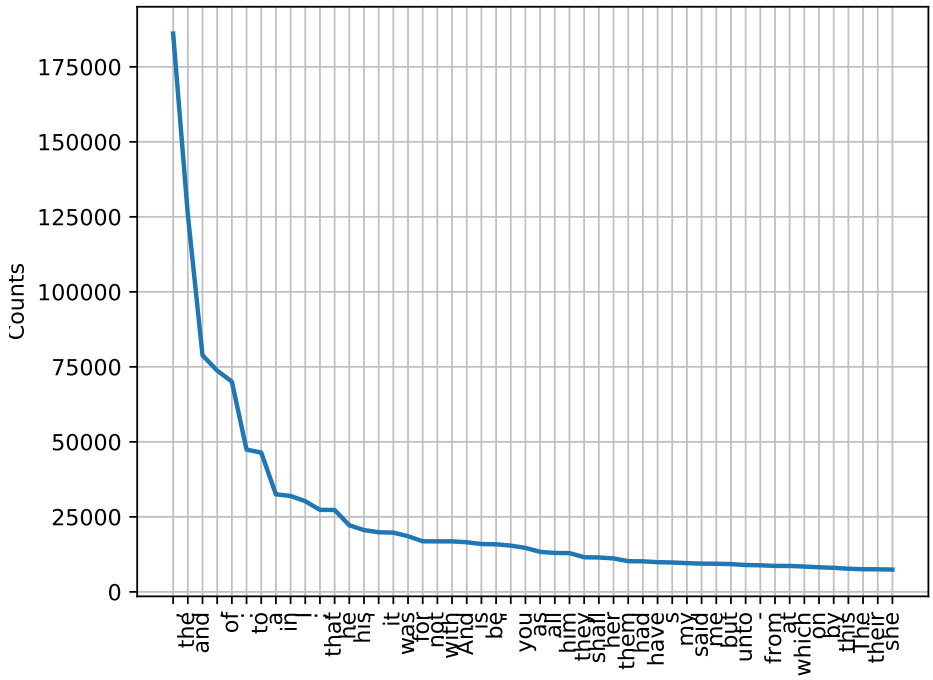

Figure 2: Frequency plot gutenberg

Removing the stop words, that represent are very much present in the top 50 results in figures 3 and 4.

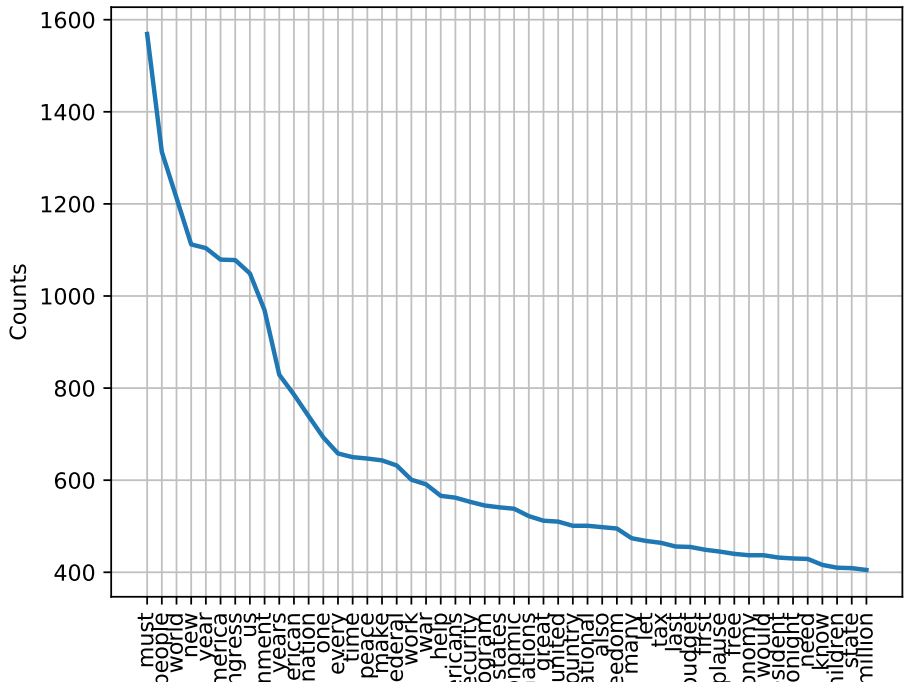

Figure 3: Frequency plot state_union

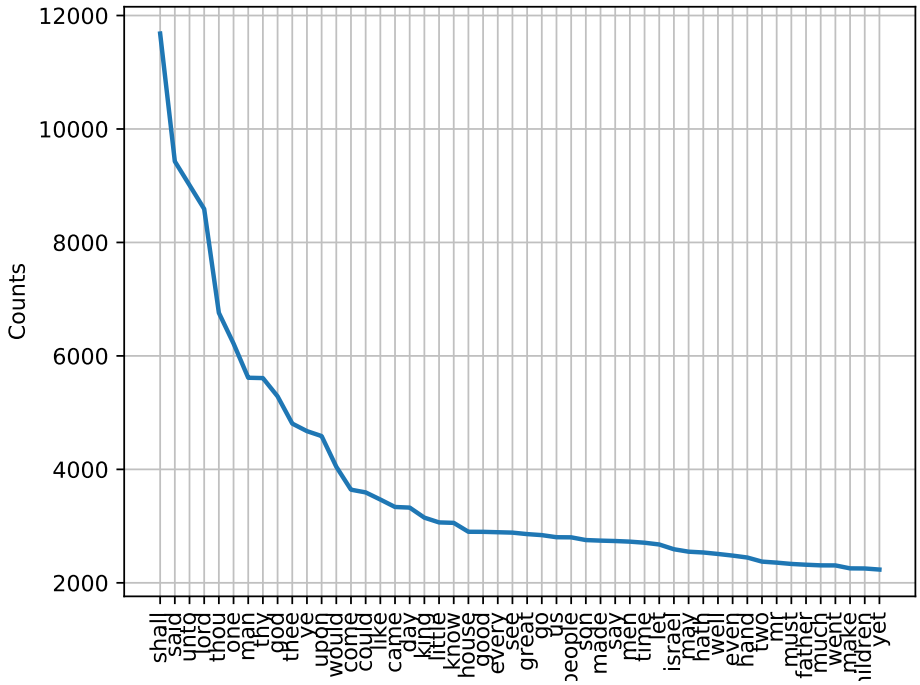

Figure 4: Frequency plot gutenberg

In table 1.1 you see all 50 listed most frequent words in state_union without dropping stopwords.

number	word	frequency
1	the	19191
2	.	17501
3	,	17484
4	of	12854
5	to	11868
6	and	11748
7	in	6936
8	a	5837
9	our	5141
10	we	4338
11	that	4309
12	for	4070
13	is	3621
14	I	3394
15	will	2959
16	’	2620
17	have	2486
18	be	2481
19	this	2323
20	are	2273
21	We	2063
22	-	1971
23	on	1857
24	with	1825
25	it	1767
26	by	1717
27	as	1663
28	all	1612
29	not	1591
30	must	1568
31	The	1520
32	–	1519
33	s	1410
34	can	1396
35	more	1369
36	people	1291
37	their	1287
38	has	1240
39	world	1128
40	year	1097
41	you	1093
42	America	1076
43	from	1058
44	And	1057
45	us	1049
46	new	1049
47	Congress	1014
48	which	1003
49	at	973
50	an	949

```
from nltk.corpus import state_union
import nltk

#remove punctuation and stopwords
all_words=(w.lower() for w in state_union.words() if (w.isalpha()) and (w.
    lower() not in nltk.corpus.stopwords.words('english')))

```

```
#Word freq count
cfd = nltk.FreqDist(all_words)

#Calculate 50 most common words
mostcommon = cfd.most_common(50)

#Plot 50 most common words and print them
cfd.plot(50)
print(mostcommon)
```

Listing 1: Code to generate term frequency with a plot

1.2 Zipf distribution

As can be seen in the figure 5 and 6 the word frequencies do follow Zipf's law. Blue is the word frequency, orange is Zipf's law.

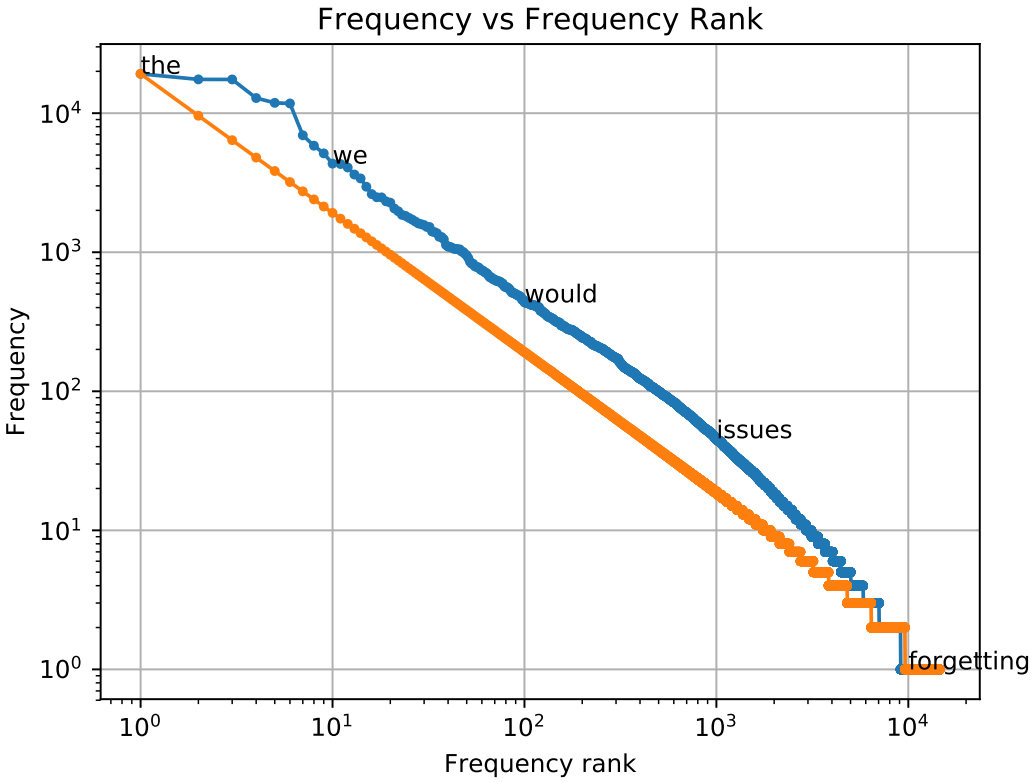

Figure 5: Zipf's law state_union

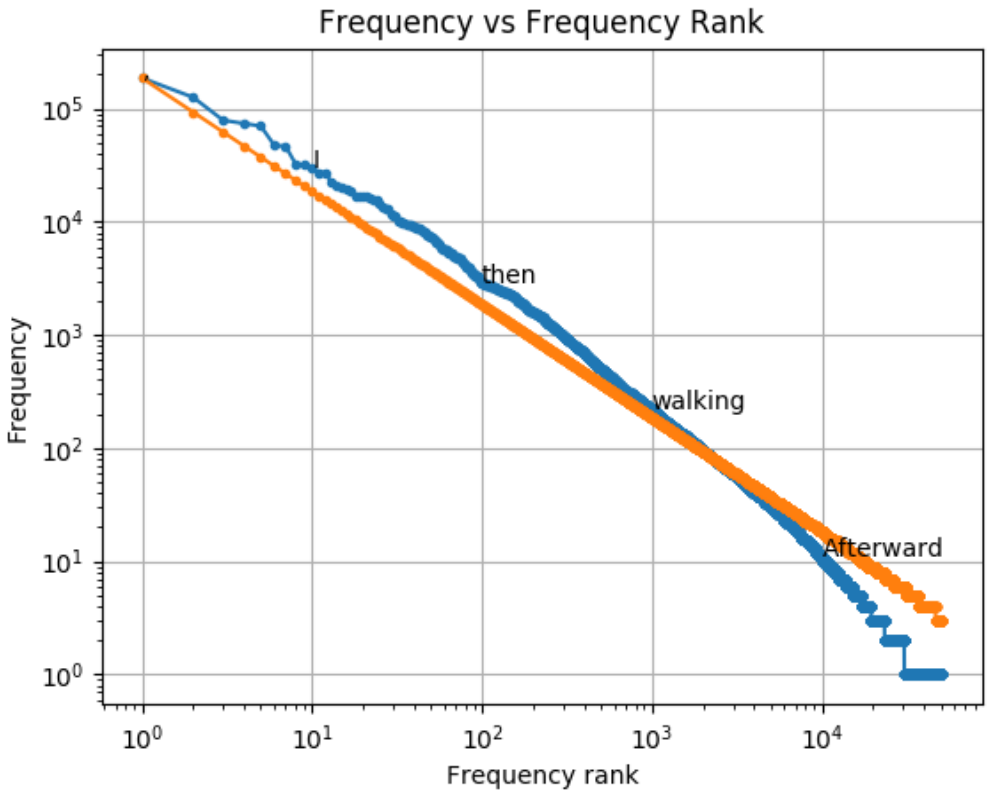

Figure 6: Zipf's law gutenberg

The figures show that Zipf's law suits gutenberg better than the state_union. Apperently state_union contains a uncommen amount of stopwords as is also shown in table 1.1. The program that created these plots can be found at listing 2.

```
import nltk
from nltk.corpus import gutenberg
from nltk.corpus import state_union
from pylab import *

def createPlot(source):
    #Word freq count
    cfd = nltk.FreqDist(source.words())

    #Add the Frequency vs Frequency Rank
    wordList = [0]
    freqList = [0]
    rankList = [0]
    curRank = 1
    for word in cfd.most_common(len(cfd.items())):
        wordList.append(word[0])
        freqList.append(word[1])
        rankList.append(curRank)
        curRank += 1
    loglog(rankList, freqList, marker=".")


    #Add the zipf function
    zipfFreqList = []
    zipfRankList = []
    for n in rankList:
        if (not (n==0)):
            zipfFreqList.append(freqList[1]/n)
```

```
            zipfRankList.append(n)
    loglog(zipfRankList, zipfFreqList, marker=".")

    grid(True)
    title("Frequency vs Frequency Rank")
    xlabel("Frequency rank")
    ylabel("Frequency")
    for n in [1,10,100,1000,10000]:
        text(rankList[n], freqList[n], wordList[n])
    show()

createPlot(state_union)
createPlot(gutenberg)
```

Listing 2: Code to plot the Zipf's distribution

1.3 Stemming and Lemmatization

In the code below we perform lemmatization , while discarding stopwords, to the state_union collection and then create a new list of the stemmed lemmas acquired.

```
import nltk
from nltk.corpus import state_union
from nltk.stem import WordNetLemmatizer
from nltk.stem.porter import PorterStemmer

#declare lematizer
lemmatizer=WordNetLemmatizer()
stemmer=PorterStemmer()

#get all non-stopwords and lemmatize them
all_lemmas =(lemmatizer.lemmatize(w.lower()) for w in state_union.words ()
    if (w.isalpha ()) and
          (w.lower() not in nltk.corpus.stopwords.words('english')))

#create empy list for stemmed lemmas
stemmed_lemmas=[]

#populate list
for word in all_lemmas:
    stemmed_lemmas.append(stemmer.stem(word))
```

Listing 3: Code to perform lemmatization in the state_union collection

1.4 10 most frequent collocations

Using the *TrigramCollocationFinder* of *nltk* we analysed the state_union text for collocations. The 10 most frequent collocations with a window size three on state_union:

1. accepts rumor gossip
2. annette strauss cochair
3. baton rouge louisiana
4. capps sonny bono
5. carrybacks recomputed amortization
6. cattle goat sheep
7. celebrates lapel pin
8. colonel qadhafi correctly

9. corporal gregory depestre
10. crust uneasily sleeping

On gutenberg this with a window of three:

1. ab origine null
2. abagtha zethar carcas
3. adalia aridatha parmashta
4. adithaim gederah gederothaim
5. agnus dei gloria
6. alammelech amad misheal
7. alang craping shtill
8. alimentive amative perceptive
9. almansor fez sus
10. amble lispe nickname

If we run the program with a larger window, for example 5 the program becomes way slower, but also the words that are present in the collocation aren't next to each other anymore. These are the timing results of running the program with different window sizes on state_union.

windowSize	time (seconds)
3	0.839669942856
5	4.19924998283
10	23.2423648834

1.5 dimensionality reduction

For this assignment we had to plot the data as a 2-dimensional scatter plot. In order to do this we had to create a scatterplot of a number of documents to spot outliers. In order to create this scatterplot we had to reduce the dimensions to two so that we were able to plot it. The choice was between SVD and Non-negative matrix factorization; where we chose for the latter. This was mainly because this seemed like a perfect fit for our case since this assumes that all values are positive which is the case for TF.IDF. Since TF.IDF is part of exercise 2 of this report we wont discuss this in this subsection. So in short we normalized the data with TF.IDF and applied dimensionality reduction with NMF. The code to do this can be seen on page 14

2 Your own search engine

2.1

For the first assignment we had to create a TF IDF representation of the document. Although the assignment states there is no built in function to do this, we did find one built into NLTK. It can be found here. However since the assignment said to create our own and some sources mentioned that the performance of this was not very good we decided to do so. To create our own TF.IDF represntation we decided to use SciKit learn. The very first step is to create a list of documents on which you want to run the TF.IDF algorithm. Next a vectorizer is called which on its turn calls a tokenizer that does some basic preprocessing like stemming and punctuation removal. This vectorizer is transformed and fit to create a TF.IDF representation. The last step is to print all of the words together with the appropriate TF.IDF scores.

The most interesting thing you can do with this is to list the top n words that do occur in one document but occur less in the other documents. However that is part of assignment two and three. So for this assignment we did some experiments with the settings of our TF.IDF vectorizer. The first thing we added were the N-Gram settings. We decided to set the n-gram range from one till three since n grams with a size of 3 seemed to be quite common and above that not very much. For the tokenizer and the stemming part we used some settings we experimented with in

exercise 1. We used a PorterStemmer, removed the punctuation, set everything to lowercase, and removed numbers. Note that the stop-word removal currently occurs twice, once as part of the vectorizer and once in the tokenize function. When we started testing our function we did not have the stop-word removal in the tokenize function and a lot of stop-words still occurred. After some tinkering it occurred to us that some stop words are not in the actual stop word list anymore after stemming. Therefore we do it twice just to be sure. The code to do this can be seen on page 14.

2.2

In this section we will discuss an interesting use of the TF.IDF representation we previously created. For every text in the Gutenberg corpus we will compute the 10 most representative words for the text. This can be done by sorting the results of the TF.IDF representation for every document and retrieving the ten highest numbers from it. To do this we used the code from the previous section and added some interesting printing. We loop over every document and for each document we get the respective row, turn it to a dense matrix, sort it and retrieve the top 10 entries. The entries are converted to the actual words and printed. The code can be seen on page 15

The results for three documents can be seen in **??** for the documents austen-sense, austen-emma, and austen-persuasion respectively. If you look at the results they make a lot of sense. A lot of names are listed in the top 10 words since those names typically do not occur in the other texts. In the third text there are even some word combinations listed like captain wentworth so the NGram option seems to be working. It would be nice if we could remove the words captain and wentworth from this list since they do not add anything. However we could not think of a way to do this without removing other words as well.

```
\label{lst:10repwords}
[ 62301 136091  47786 187082 115337  70432 239964  61255 199034 132015]
elinor
mariann
dashwood
said
jen
everi
willoughbi
edward
sister
luci
------------------------next doc
[ 65218  97001  64434 118468 142512 238169 242629 187082 219372 218234]
emma
harriet
elton
knightley
miss
weston
woodhous
said
think
thing
------------------------next doc
[146324  63594   9435  28157 237744  28265 146457 147007  31783 186447]
mr
elliot
ann
captain
wentworth
captain wentworth
mr elliot
musgrov
charl
russel
------------------------next doc
```

2.3

In this section we discuss the function **where_was_that** which we created. As arguments it takes a string to search for and a corpus to search in. The main job of the function is to search through all of the documents trying to figure out which text it belong to. Note that it does not have to be an actual sentence from the text but it can also be a sentence describing the text.

The main idea of our program is that we create a TF.IDF representation of all of the documents in the provided corpus. For every document we check the number of matches from the provided sentence and the top N words from the TF.IDF representation. The document with the most matches is the most likely to be the document searched for.

The code for this program can be found on page 16. The function **get_corpus_rep_words** return the top N TF.IDF words for every document in a 2-dimensional list. In the **where_was_that** function a loop is started where using the **&** operator the number of matches between the sentence and the TF.IDF representation is returned. The document number, number of matches and document name are put into tuples which are appended to a list. When the for loop is done the list is sorted based on the number matches and the first 10 results are returned.

The main feature we experimented with in this case was what number for **top_n**. At first we configured this to be the entire length of the text however that does not make any sense since then you would just search through a list of words and count the matches. However it does produces correct results for both samples.

To make it more interesting we currently set the number top_n to 1000. This gives correct results for the Moby Dick sample phrase but not for the Alice in Wonderland sample which comes in shared second. However the number of words to look through is hugely reduced which is very nice. You could for example create a TF.IDF representation of a number of corpus once after which you only have to look through 1000 words for every document compared to about 100 000 for every document resulting in a nice speed improvement.

For reference we created **??** with the output for both examples with different top_n values. As you can see the output for Moby Dick is always correct (even with n_top of 1!). However Alice in wonderland is only correct for n_top of 100 where it is partially first.

```
\label{lst:wherewasthatres}
10000  textWonderland
[(5, 'bryant-stories.txt', 5), (8, 'chesterton-ball.txt', 5), (9, '
    chesterton-brown.txt', 5), (10, 'chesterton-thursday.txt', 5), (7, '
    carroll-alice.txt', 4), (11, 'edgeworth-parents.txt', 4), (12, '
    melville-moby_dick.txt', 4), (17, 'whitman-leaves.txt', 4), (0, 'austen
    -emma.txt', 3), (1, 'austen-persuasion.txt', 3)]

10000  textMoby
[(8, 'chesterton-ball.txt', 3), (12, 'melville-moby_dick.txt', 3), (17, '
    whitman-leaves.txt', 3), (1, 'austen-persuasion.txt', 2), (5, 'bryant-
    stories.txt', 2), (9, 'chesterton-brown.txt', 2), (10, 'chesterton-
    thursday.txt', 2), (11, 'edgeworth-parents.txt', 2), (0, 'austen-emma.
    txt', 1), (2, 'austen-sense.txt', 1)]

1000   textWonderland
[(5, 'bryant-stories.txt', 4), (7, 'carroll-alice.txt', 4), (9, 'chesterton
    -brown.txt', 4), (8, 'chesterton-ball.txt', 3), (11, 'edgeworth-parents
    .txt', 3), (12, 'melville-moby_dick.txt', 3), (0, 'austen-emma.txt', 2)
    , (4, 'blake-poems.txt', 2), (10, 'chesterton-thursday.txt', 2), (17, '
    whitman-leaves.txt', 2)]

1000 textMoby
[(12, 'melville-moby_dick.txt', 3), (5, 'bryant-stories.txt', 2), (1, '
    austen-persuasion.txt', 1), (7, 'carroll-alice.txt', 1), (9, '
    chesterton-brown.txt', 1), (11, 'edgeworth-parents.txt', 1), (17, '
    whitman-leaves.txt', 1), (0, 'austen-emma.txt', 0), (2, 'austen-sense.
    txt', 0), (3, 'bible-kjv.txt', 0)]

100  textWonderland
[(5, 'bryant-stories.txt', 1), (7, 'carroll-alice.txt', 1), (0, 'austen-
    emma.txt', 0), (1, 'austen-persuasion.txt', 0), (2, 'austen-sense.txt',
```

```
 0), (3, 'bible-kjv.txt', 0), (4, 'blake-poems.txt', 0), (6, 'burgess-
    busterbrown.txt', 0), (8, 'chesterton-ball.txt', 0), (9, 'chesterton-
    brown.txt', 0)]

100 textMoby
[(12, 'melville-moby_dick.txt', 2), (5, 'bryant-stories.txt', 1), (0, '
    austen-emma.txt', 0), (1, 'austen-persuasion.txt', 0), (2, 'austen-
    sense.txt', 0), (3, 'bible-kjv.txt', 0), (4, 'blake-poems.txt', 0), (6,
     'burgess-busterbrown.txt', 0), (7, 'carroll-alice.txt', 0), (8, '
    chesterton-ball.txt', 0)]

10 textWonderland
[(0, 'austen-emma.txt', 0), (1, 'austen-persuasion.txt', 0), (2, 'austen-
    sense.txt', 0), (3, 'bible-kjv.txt', 0), (4, 'blake-poems.txt', 0), (5,
     'bryant-stories.txt', 0), (6, 'burgess-busterbrown.txt', 0), (7, '
    carroll-alice.txt', 0), (8, 'chesterton-ball.txt', 0), (9, 'chesterton-
    brown.txt', 0)]

10 textMoby
[(12, 'melville-moby_dick.txt', 1), (0, 'austen-emma.txt', 0), (1, 'austen-
    persuasion.txt', 0), (2, 'austen-sense.txt', 0), (3, 'bible-kjv.txt',
    0), (4, 'blake-poems.txt', 0), (5, 'bryant-stories.txt', 0), (6, '
    burgess-busterbrown.txt', 0), (7, 'carroll-alice.txt', 0), (8, '
    chesterton-ball.txt', 0)]


1 textWonderland
[(0, 'austen-emma.txt', 0), (1, 'austen-persuasion.txt', 0), (2, 'austen-
    sense.txt', 0), (3, 'bible-kjv.txt', 0), (4, 'blake-poems.txt', 0), (5,
     'bryant-stories.txt', 0), (6, 'burgess-busterbrown.txt', 0), (7, '
    carroll-alice.txt', 0), (8, 'chesterton-ball.txt', 0), (9, 'chesterton-
    brown.txt', 0)]

1 textMoby
[(12, 'melville-moby_dick.txt', 1), (0, 'austen-emma.txt', 0), (1, 'austen-
    persuasion.txt', 0), (2, 'austen-sense.txt', 0), (3, 'bible-kjv.txt',
    0), (4, 'blake-poems.txt', 0), (5, 'bryant-stories.txt', 0), (6, '
    burgess-busterbrown.txt', 0), (7, 'carroll-alice.txt', 0), (8, '
    chesterton-ball.txt', 0)]
```

In our program most time is spent on the stopword removal in the tokenize function however we could not think of a way to improve this performance. This is not a very big problem however in the case described above where the TF.IDF representation is created only once and stored. Something which could be improved is the way the best matching document is found. Currently matching is done using a counter for every matched word which makes the chance that multiple documents end up as possible candidates quite large. You could add up the TF.IDF scores for all words in the given sentence and choose the document resulting in the highest score. The last thing we did some experiments with was creating a TF.IDF representation of the sentence itself as well. The first thing we tried was putting a single sentence in the vectorizer. We were not sure how it computed the IDF component since there were no other documents but it did produce some results. After retrieving the most important words from the sentence we tried searching for them in the TF.IDF model however this did not produce very good results. The next thing was to include the sentence in the TF.IDF model created from the corpus however this did not seem to improve the results. This might be due to the stop-words already being removed from the sentence and the other words are already representative for the text it describes.

So in short; we are able to find the correct text for both samples if we search through the entire text however that is not a very interesting approach. When we search through the top 1000 words for every document we do find the correct text for Moby Dick but not for Alice in Wonderland (although ranked high). This however is an interesting approach since you use a model which has to be created once after which you only have to search through about 1% of the original text. We did try to do some experiments on how it works however none of the experiments resulted in a better performance.

3 Appendix

1.5

```
# -*- coding: utf-8 -*-
from nltk.corpus import gutenberg
import nltk
from nltk.corpus import stopwords
from sklearn.decomposition import NMF

from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.stem.porter import PorterStemmer

token_dict = []
stemmer = PorterStemmer()

def stem_tokens(tokens, stemmer):
    stemmed = []
    for item in tokens:
        stemmed.append(stemmer.stem(item))
    return stemmed

def tokenize(text):
    tokens = nltk.word_tokenize(text)
    cleantext = (w for w in tokens if (w.isalpha() and w not in stopwords.
        words('english')))
    stems = stem_tokens(cleantext, stemmer)
    return stems



#texts = gutenberg.fileids()
#for text in texts:
#    token_dict.append(gutenberg.raw(text))
#    break

#Three manually added texts if needed
token_dict.append(gutenberg.raw('austen-sense.txt'))
token_dict.append(gutenberg.raw('austen-emma.txt'))
token_dict.append(gutenberg.raw('austen-persuasion.txt'))

print("Docs added!")

tfidf = TfidfVectorizer(tokenizer=tokenize, stop_words='english', lowercase
    = True, ngram_range=(1,3))
print("Vectorizer created!")
tfs = tfidf.fit_transform(token_dict)
print("Model fit!")

#tfidf = vectorizer.fit_transform(dataset.data)

nmf = NMF(n_components=2, random_state=1)
X = nmf.fit_transform(tfs)
print("NMF done!")
```

Listing 4: Normalization with TF.IDF; Dimensionality Reduction with NMF

2.1

```
from nltk.corpus import gutenberg
import nltk
from nltk.corpus import stopwords
```

```

from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.stem.porter import PorterStemmer

token_dict = []
stemmer = PorterStemmer()

def stem_tokens(tokens, stemmer):
    stemmed = []
    for item in tokens:
        stemmed.append(stemmer.stem(item))
    return stemmed

def tokenize(text):
    tokens = nltk.word_tokenize(text)
    cleantext = (w for w in tokens if (w.isalpha() and w not in stopwords.
        words('english')))
    stems = stem_tokens(cleantext, stemmer)
    return stems


texts = gutenberg.fileids()
for text in texts:
    token_dict.append(gutenberg.raw(text))

#this can take some time
tfidf = TfidfVectorizer(tokenizer=tokenize, stop_words='english', lowercase
    = True, ngram_range=(1,3))
tfs = tfidf.fit_transform(token_dict)

#Print the word names with tf-idf score
feature_names = tfidf.get_feature_names()
for col in tfs.nonzero()[1]:
    print(feature_names[col], ' - ', tfs[0, col])
```

Listing 5: Code to generate tfidf scores

2.2

```

from nltk.corpus import gutenberg
import nltk
import numpy as np
from nltk.corpus import stopwords

from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.stem.porter import PorterStemmer

token_dict = []
stemmer = PorterStemmer()

def stem_tokens(tokens, stemmer):
    stemmed = []
    for item in tokens:
        stemmed.append(stemmer.stem(item))
    return stemmed

def tokenize(text):
    tokens = nltk.word_tokenize(text)
    cleantext = (w for w in tokens if (w.isalpha() and w not in stopwords.
        words('english')))
    stems = stem_tokens(cleantext, stemmer)
    return stems

```

```
24
25 #Append all gutenberg texts to dictionairy
26 ##for fileid in gutenberg.fileids():
27 #     token_dict.append(gutenberg.raw(fileid))
28
29 #Three manually added texts if needed
30 token_dict.append(gutenberg.raw('austen-sense.txt'))
31 token_dict.append(gutenberg.raw('austen-emma.txt'))
32 token_dict.append(gutenberg.raw('austen-persuasion.txt'))
33
34 #Create TFIDF model
35 tfidf = TfidfVectorizer(tokenizer=tokenize, stop_words='english', lowercase
       = True, ngram_range=(1,3))
36 tfs = tfidf.fit_transform(token_dict)
37 idf = tfidf._tfidf.idf_
38 feature_names = np.array(tfidf.get_feature_names())
39
40 #Calculate the top n words for every document
41 top_n = 10
42 for i in range(len(token_dict)):
43     wordindexes =  tfs.getrow(i).todense().A1.argsort()[-top_n:][::-1]
44     print (wordindexes)
45     wordfeatures = tfidf.get_feature_names()
46     for i in wordindexes:
47         print (wordfeatures[i])
48     print ("------------------------next doc")
```

Listing 6: Code to generate most representative words for each text in Gutenberg

2.3

```
1  # -*- coding: utf-8 -*-
2  from nltk.corpus import gutenberg
3  import nltk
4  from nltk.corpus import stopwords
5
6  from sklearn.feature_extraction.text import TfidfVectorizer
7  from nltk.stem.porter import PorterStemmer
8
9  stemmer = PorterStemmer()
10
11 # Function to do stemming
12 def stem_tokens(tokens, stemmer):
13     stemmed = []
14     for item in tokens:
15         stemmed.append(stemmer.stem(item))
16     return stemmed
17
18 # Tokenizer removing punctuation and remoivng capitals
19 def tokenize(text):
20     tokens = nltk.word_tokenize(text)
21     cleantext = (w for w in tokens if (w.isalpha() and w not in stopwords.
           words('english')))
22     stems = stem_tokens(cleantext, stemmer)
23     return stems
24
25
26 # Function calculating the tfidf rating for every document
27 def get_corpus_rep_words(corpus, top_n):
28     #Append all gutenberg texts to dictionairy
29     token_dict = []
30     repwords = []
31     for fileid in corpus.fileids():
```

```
        token_dict.append(corpus.raw(fileid))

    #Create TFIDF model
    tfidf = TfidfVectorizer(tokenizer=tokenize, stop_words='english',
        lowercase = True, ngram_range=(1,3))
    tfs = tfidf.fit_transform(token_dict)

    #Calculate the top n words for every document
    for j in range(len(token_dict)):
        wordindexes =  tfs.getrow(j).todense().A1.argsort()[-top_n:][::-1]
        wordfeatures = tfidf.get_feature_names()
        templist = []
        for i in wordindexes:
            templist.append(wordfeatures[i])
        repwords.append(templist)
    return repwords

# Converting a index to the docuement name
def get_doc_from_IDX(index, corpus):
   docList = corpus.fileids()
   return docList[index]

# Where was that function from assignement
def where_was_that(text, corpus):
    repWords = get_corpus_rep_words(corpus, 10000)
    stems = tokenize(text)
    matches = []
    for idx, doc in enumerate(repWords):
        # Add a tuple to list consisting of (document index, document name,
            number of matches)
        matches.append((idx, get_doc_from_IDX(idx, corpus), (len(set(doc) &
            set(stems)))))

    # Sort based on number of matches
    sortedList = sorted(matches, key=lambda tup: tup[2], reverse=True)
    return(sortedList[:10])


# Sample texts from assignement
textWonderland = "story with the girl falling into a rabbit hole"
textMoby = "Story with the sailor and the whale"
print(where_was_that(textWonderland, gutenberg))
```

Listing 7: Search engine program that searches through a corpus based on provided text and returns the document that is most likely to contain it.