NLP PROJECT REPORT

Team: Sic Mundus Creatus Est.

Team Members:

- Chhavi Batra (18ucs192)
- Dhruv Pandya (18ucs007)
- Deepanshu Somani (18ucs105)

Github Project link:

https://github.com/SIcMundusCreatus/Final_Project

DATA DESCRIPTION:

The two texts chosen by the team are as follows:

T1: **"The Hounds of Baskerville"**, by Sir Arthur Conan Doyle

T2: **"Pride and Prejudice"**, by Jane Austen

Both the texts are in .txt format and in 'utf-8' encoding. The texts are opened by using the URLs linking to the texts. URLs of the text are:

T1: http://gutenberg.org/files/2852/2852-0.txt

T2: http://gutenberg.org/files/1342/1342-0.txt

We have used the ***urllib.request.urlopen(url)*** function to read the URL files into variables named t1 and t2 respectively for both the books.

The data type of both the texts comes out to be 'bytes', which should be converted into string (str) format to preprocess the data. This can be achieved by using the ***str.decode(encoding)*** method supported by the string class to decode any string. The encoding is passed into the parameter encoding of the function(utf-8 in our case).

After decoding the resultant texts in string(str) format are assigned to variables rawbook1 and rawbook2 respectively. Below is the snippet of code where this can be seen:

```
rawbook1 = t1.decode('utf-8')
rawbook2 = t2.decode('utf-8')
```

```
[ ] print(type(rawbook1))
    print(type(rawbook2))
```

```
<class 'str'>
<class 'str'>
```

DATA PREPROCESSING STEPS:

Data preprocessing mainly consists of cleaning the data. This involves removing the lines of text from the text string which do not contribute to the data analysis. We only need to have meaningful text which is the content of each chapter. We can remove the starting and ending lines of the text by carefully removing unnecessary text lines.

We can divide the whole text string into a list of string that contains every line from the initial text string. For this, we use the ***str.splitlines()*** method from the str class. The function divides the lines based on the fact that the '\n' character is used to represent a new line in the text string.

The result is assigned to the variables *rawbook1_line_list* and *rawbook2_line_list* respectively. Both of the variables contain a list of every line.

Then we calculate the starting index of the chapters of both the books. Below are the code snippets which show the logic we have used to achieve that :

```
start_index = 0
end_index = 0

for line in rawbook1_line_list[0:]:
  if line == "Chapter 1.":
     print(start_index)
  if line == "THE END":
     print(end_index)
     break
  start_index += 1
  end_index += 1
```

```
89
7370
```

The above image is for the first book. In the book, "The Hound of Baskervilles", we see that the useful content starts at the line "Chapter 1." which is different from the notation used in the contents table for the book. The end of the novel is marked by the sentence "THE END". All the lines after this line are useless for the data analysis that we have to perform. Thus using the above code we can find the useful part of the book.

For the second book, a similar approach can be taken. The code snippet is given below:

```
start_index = 0;
end_index = 0;
for line in rawbook2_line_list[0:]:
  if line == "      Chapter 1":
     print(start_index)
  if line == "End of the Project Gutenberg EBook of Pride and Prejudice, by Jane Austen":
     print(end_index)
     break
  start_index += 1
  end_index +=1
```

```
167
14231
```

Data Cleaning for Book 1 (T1)

Based on the start index and the end index obtained for *rawbook1_list_line*, we slice the list so that only the chapter names and the contents of the chapter remain. This new list is assigned to the variable named *raw_list1.* We join the remaining list of string to form the new reduced text string, which is assigned to the variable *joined_book1*.

To extract the names of all the chapters, we use the content table of the book. Using regular expressions we can extract the chapter number followed by the chapter name and convert this line into the pattern in which the chapter number and chapter name appear in the remaining text. The pattern that the chapter number and the chapter name appear before every chapter is “Chapter i.ChapterName”, where i is the chapter number.

We first use the regular expressions to make a list of strings for each chapter in the above-mentioned format. To do this we have used the following code :

```
import regex as re
chapter_list = []
pattern = r' Chapter [0-9][0-9]?'
for line in rawbook1_line_list[:88]:
   if bool(re.search(pattern, line)):
     line = re.sub("\t",".",line)
     temp_line = line[1:]
     chapter_list.append(temp_line)
print(chapter_list)
```

```
['Chapter 1.Mr. Sherlock Holmes', 'Chapter 2.The Cu
```

We have used the ***re.sub()*** method available in the regex module to replace the tab space present between the chapter number and chapter name in the content list of the book. The first string of the chapter_list variable can be seen in the image above.

Using the variable *chapter_list*, we can remove the chapter number and chapter name for each chapter. The code is:

```
for item in chapter_list:
  joined_book1 = re.sub(item,"",joined_book1)
joined_book1 = joined_book1[1:]
joined_book1
```

Thus the variable *joined_book1* finally contains only the part of the text string required for data analysis.

Data Cleaning for Book 2 (T2)

As done above for book 1, we slice the list *rawbook2_list_line* using the *start_index* and *end_index*
obtained as mentioned in the code snippet above. This new list after the removal of unwanted text is stored in the variable *raw_list2.* We join the remaining list of string to form the new reduced text string, which is assigned to the variable *joined_book2*. The string joined_book2 is further processed by replacing the tab spaces with the single spaces using the ***replace()*** function as shown in the code snippet below.

```
[ ]  raw_list2 = rawbook2_line_list[167:14227]
     joined_book2 = ''.join(raw_list2)
     joined_book2 = joined_book2.replace('      ',' ')
     joined_book2
```

Now we extract the Chapter headings like "Chapter i" present in the book T2 before starting of the every ith chapter. This is achieved by using ***sub(old,new,string)*** function, where a regular expression corresponding to the pattern of chapter heading, stored in the variable *pattern* has been passed as the old parameter. In the new parameter, we pass an empty string, so that all the chapter headings are removed from the text T2. This is done as follows:

```
[ ]  import regex as re

     pattern = r'Chapter [0-9][0-9]?'
     joined_book2 = re.sub(pattern,"",joined_book2)
     joined_book2 = joined_book2[2:]
     joined_book2
```

Thus the variable *joined_book1* finally contains only the part of the text string required for data analysis

DATA PREPARATION STEPS:

TOKENIZATION

In the data preparation steps, we perform tokenization so as to prepare the texts T1, T2 data for analysis of word length and frequency, and for Data Cloud visualization.

We achieve the task of extracting tokens from the strings T1 and T2 by using the ***nltk.word_tokenize()*** method. The function ***word_tokenize()*** provided by **NLTK** splits the strings into tokens on the basis of white spaces and punctuation. So we pass the strings *joined_book1* and *joined_book2* in the function and obtain the list of tokens for T1 and T2 which is stored in the variables *tokens_t1* and *tokens_2* respectively as shown below.

```
[ ]  tokens_t1 = nltk.word_tokenize(joined_book1)
     tokens_t2 = nltk.word_tokenize(joined_book2)
```

The first ten tokens in the lists *tokens_t1* and *tokens_t2* corresponding to the texts T1 and T2 respectively are shown below:

```
tokens_t1[:10]
```

```
['Mr.',
 'Sherlock',
 'Holmes',
 ',',
 'who',
 'was',
 'usually',
 'very',
 'late',
 'in']
```

```
tokens_t2[:10]
```

```
['it',
 'is',
 'a',
 'truth',
 'universally',
 'acknowledged',
 ',',
 'that',
 'a',
 'single']
```

Problem Statement:

In this project following tasks have been performed the and then respective conclusions and inferences have been drawn:

- Analyzing the frequency distribution of tokens in T1 and T2 separately, and then drawing out conclusions for both the distributions.
- Analyzing the Word Cloud before and after the removal of stop words for both T1 and T2 separately.
- Evaluating the relationship between the word length and frequency for both T1 and T2 and stating the necessary result.
- PoS Tagging for both T1 and T2.

Analysis of frequency distribution of tokens

To analyze the frequency distribution of tokens, we need to calculate the counts of unique values for each of the texts. We first convert the list of tokens *tokens_t1 and tokens_t2* into a series, which is a 1-D labeled array data structure of Panda library.

```
[23] import pandas as pd
```

```
[24] token_series_t1 = pd.Series(tokens_t1)
     token_series_t2 = pd.Series(tokens_t2)
```

We then use ***Series.value_counts()*** function to return a series that contains counts of unique values of tokens for T1 and T2. The frequency distribution series for T1 and T2 are stored in the variables *token_series_t1* and *token_series_t2* respectively as shown in the code snippet below.

Frequency distribution of T1

```
[ ] freq_dist_t1 = token_series_t1.value_counts()
    print(type(freq_dist_t1))
    freq_dist_t1
```

```
<class 'pandas.core.series.Series'>
,              3433
the            3057
.              2499
of             1581
and            1505
               ...
recruited         1
Marcini           1
unfair            1
winds             1
expeditions       1
Length: 6504, dtype: int64
```

We can see from the above snippet the frequency distribution in the descending order of the frequency of tokens. And we can see that the most occurring token **'the'** is succeeded by the punctuation **','.** We have included the punctuations as well because the function

word_tokenize() provided by **NLTK** splits the strings into tokens on the basis of white spaces and punctuation.

Frequency distribution of T2

```
freq_dist_t2 = token_series_t2.value_counts()
print(type(freq_dist_t2))
freq_dist_t2
```

```
<class 'pandas.core.series.Series'>
,           9131
to          4075
.           4075
the         4048
of          3582
            ...
thwarted       1
sequel         1
unheard        1
reckon         1
acutest        1
Length: 7725, dtype: int64
```

Similar to T1 is the frequency distribution of **T2**. And we can observe that here the most occurring token is **“to”**, which is again succeeded by **“,”** as in T1.

WordCloud analysis before and after removal of stop words

It is a data visualization technique used for representing text data where each word indicates its frequency. So large is the size of the word, more is the frequency.

So, for generating word cloud of T1 and T2 we need to import method ***WordCloud*** and pyplot method from the library ***matplotlib*** which can be used to plot a word cloud***.***

```
[ ]  from wordcloud import WordCloud
     import matplotlib.pyplot as plt
```

STOPWORDS: These are the words that are most common in a language. For example: and the, etc. Stopwords occur very frequently in a text.

Text T1’s WordCloud

We have taken a variable *comment_words_t1* which is an empty string. After that, we have applied a loop in which first we have lowercased all the words and then joined them in *comment_words_t1* with space before and after.

```
[ ]  comment_words_t1 = ''
     for i in range(len(tokens_t1)):
       tokens_t1[i] = tokens_t1[i].lower()

     comment_words_t1 += " ".join(tokens_t1)+" "
```

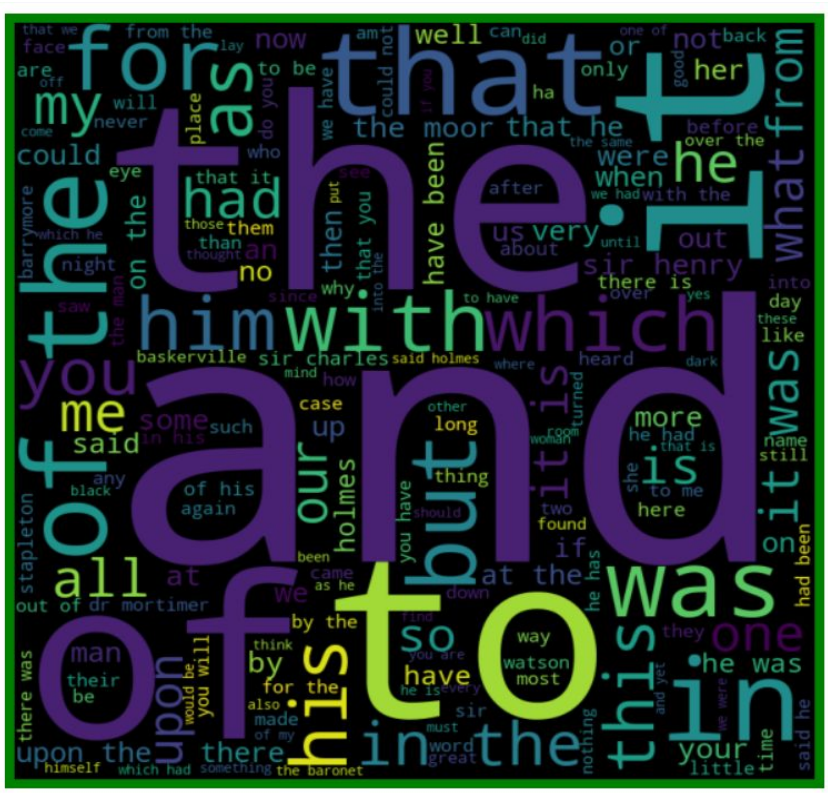
As we already have imported the ***WordCloud*** method from **word cloud** library.
So we will directly use this method to generate a word cloud and pass various parameters according to our choice and the word cloud is generated.
Here we are taking stopwords as ',' because in WordCloud function the stopwords are automatically embedded (parameter is already passed) and it automatically removes the stopwords from the word cloud and we dont want that.So we have considered a single character i.e. ','

```
wordcloud_t1 = WordCloud(width = 800, height = 800,
                background_color ='black',
                min_font_size = 10 , stopwords={','}).generate(comment_words_t1)
```

Using the ***matplotlib*** library, we will show the word cloud according to the code:

```
plt.figure(figsize = (8, 8), facecolor = 'green')
plt.imshow(wordcloud_t1)
plt.axis("off")
plt.tight_layout(pad = 0)
plt.show()
```

Word cloud for T1:



Text T1's Wordcloud without StopWords

Firstly, we will download the stopwords from **nltk** library and then store it in variable ***all_stopwords*** for the English language.

```
[ ] nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
True
```

```
[ ] from nltk.corpus import stopwords
    all_stopwords = stopwords.words('english')
```

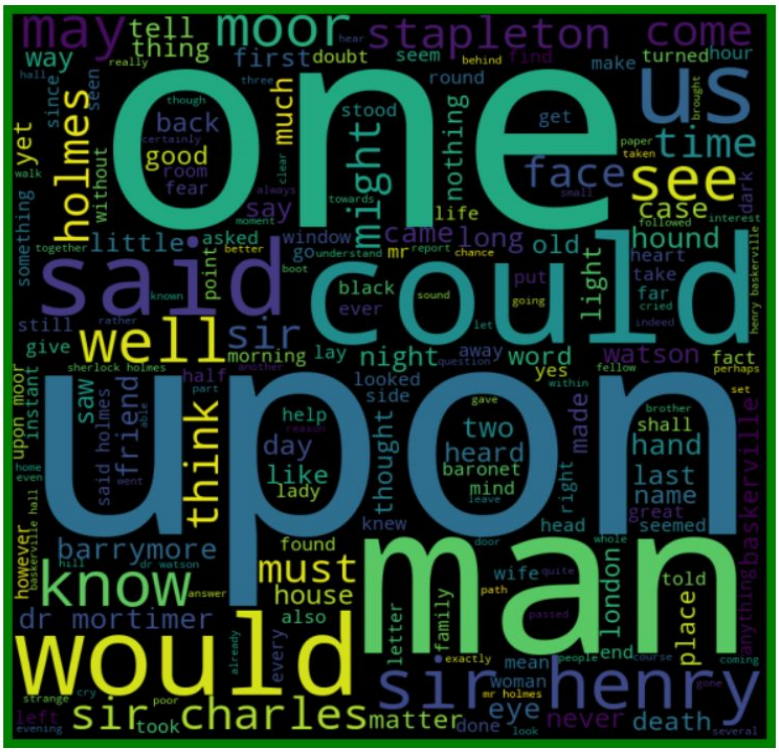
Now, we will apply the ***WordCloud*** function again but this time we are going to include the stop words stored in *all_stopwords* and all the other parameters are included as well.

```
[ ] wordcloud_sw_t1 = WordCloud(width = 800, height = 800,
                    background_color ='black',
                    stopwords = all_stopwords,
                    min_font_size = 10 ,).generate(comment_words_t1)
```

And now show the generated word cloud by ***matplotlib***

```
plt.figure(figsize = (8, 8), facecolor = 'green')
plt.imshow(wordcloud_sw_t1)
plt.axis("off")
plt.tight_layout(pad = 0)
plt.show()
```

Word Cloud for T1 without Stop Words:



Text T2's WordCloud

Here we will follow steps similar to the ones followed in T1's word cloud. The code snippets for T2 can be seen below :

```
comment_words_t2 = ''
for i in range(len(tokens_t2)):
  tokens_t2[i] = tokens_t2[i].lower()

comment_words_t2 += " ".join(tokens_t2)+" "
```

```
wordcloud_t2 = WordCloud(width = 800, height = 800,
                background_color ='black',
                min_font_size = 10 , stopwords={','}).generate(comment_words_t2)
```

```
[ ]  plt.figure(figsize = (8, 8), facecolor = 'green')
     plt.imshow(wordcloud_t2)
     plt.axis("off")
     plt.tight_layout(pad = 0)
     plt.show()
```

Word Cloud for T2 including Stopwords:

Text T2's WordCloud after removing stop words

We will follow the same steps followed in T1's Word cloud:

```
[ ]  wordcloud_sw_t2 = WordCloud(width = 800, height = 800,
                    background_color ='black',
                    stopwords = all_stopwords,
                    min_font_size = 10 ,).generate(comment_words_t2)
```

```
plt.figure(figsize = (8, 8), facecolor = 'green')
plt.imshow(wordcloud_sw_t2)
plt.axis("off")
plt.tight_layout(pad = 0)
plt.show()
```

Word Cloud for T2:

Inference of Word Clouds

The difference between the word cloud before removing the stopwords and after removing the stop words can be seen clearly as the common words like and, the, etc have been removed and the remaining words have been considered for forming the word cloud.

In the two word clouds for the first book, we can observe that the first word cloud that includes the stopwords has some of the stopwords as the larger represented words in the figure. For the second word cloud which does not include the stop words, words like upon, could, etc. are more prominent.

In the two word clouds for the second book, we can observe that the first word cloud that includes the stopwords has some of the stopwords as the larger represented words in the figure. For the second word cloud which does not include the stop words, words like sister, must, etc. are more prominent. Even Elizabeth, which is the name of one of the characters, is more prominent.

Pos Tagging

It is the process of assigning one of the parts of speech to the given word.
Firstly, we have downloaded the **universal_tagset** from the **nltk** library.

```
nltk.download('universal_tagset')
```

```
[nltk_data] Downloading package universal_tagset to /root/nltk_data...
[nltk_data]   Unzipping taggers/universal_tagset.zip.
True
```

For T1

Now, we will import the pos_tag function from nltk library to tag our text words. This uses averaged_perceptron_tagger.

And then download averaged_perceptron_tagger which contains the pre-trained English POS tagging. Averaged_perceptron_tagger is developed by Matthew Honnibal. He has defined it to have similarities with a trigram model of the Viterbi Algorithm. The averaged_perceptron_tagger tags the word i by looking at the tag of the previous word i-1 and the three last letters of the following word (i+1th word).

Finally, using method pos_tag and passing the parameter as tokens_t1 and storing it in nltk_data_t1.

```
[50] from nltk import pos_tag
     nltk.download('averaged_perceptron_tagger')
     nltk_data1 = pos_tag(tokens_t1)
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
```

```
for i in nltk_data1[:]:
  print(i)
```

For T2

Similarly, we will do it for T2 but the parameter for ***pos_tag*** will be *tokens_t2.*

```
[52] nltk_data2 = pos_tag(tokens_t2)
```

```
for i in nltk_data2[:]:
  print(i)
```

Inference from POS Tagging

For most of the words, the POS tags are appropriate. There are some instances where the tagging is wrong. The tagger used uses weighted tagging. The punctuation marks are also tagged but there are many discrepancies.

For the text T1, some of the tags observed are:

```
('mr.', 'NN')
('sherlock', 'NN')
('holmes', 'NNS')
(',', ',')
('who', 'WP')
('was', 'VBD')
('usually', 'RB')
('very', 'RB')
('late', 'RB')
('in', 'IN')
('the', 'DT')
('mornings', 'NNS')
```

For the text T2, tags are:

('it', 'PRP')
('is', 'VBZ')
('a', 'DT')
('truth', 'NN')
('universally', 'RB')
('acknowledged', 'VBD')
(',', ',')
('that', 'IN')
('a', 'DT')
('single', 'JJ')
('man', 'NN')
('in', 'IN')
('possession', 'NN')

Analysis of Word Length and Frequency

A suitable measure of analysis of the distribution of tokens for a large text can be the analysis of word length with the frequency of each word length. This can be done by making a dictionary that maps the length of the word to the number of words with the given length.

For Text T1

For the text t1, we can use the token list *tokens_t1*, prepared earlier to save all the tokens in the cleaned version of the text which was saved in the variable *joined_book*. We can use this token list to create a dictionary *wordlen_freq_t1,* which represents the dictionary format discussed above.

This can be created by simple code logic as shown below:

```
wordlen_freq_t1 = {}
for item in tokens_t1:
  length = len(item)
  if(length == 18):
    print("18 length word : " + str(item))
  if(length == 23):
    print("23 length word : " + str(item))

  if length in wordlen_freq_t1:
    wordlen_freq_t1[length] = wordlen_freq_t1[length] + 1
  else:
    wordlen_freq_t1[length] = 1
```

```
18 length word : remarkable-looking
23 length word : self-respect—everything
18 length word : questioning—stared
18 length word : ill-treatment—that
```

We have used two if the statements in the code to print a few comparatively longer words which have a length 18 and 23.

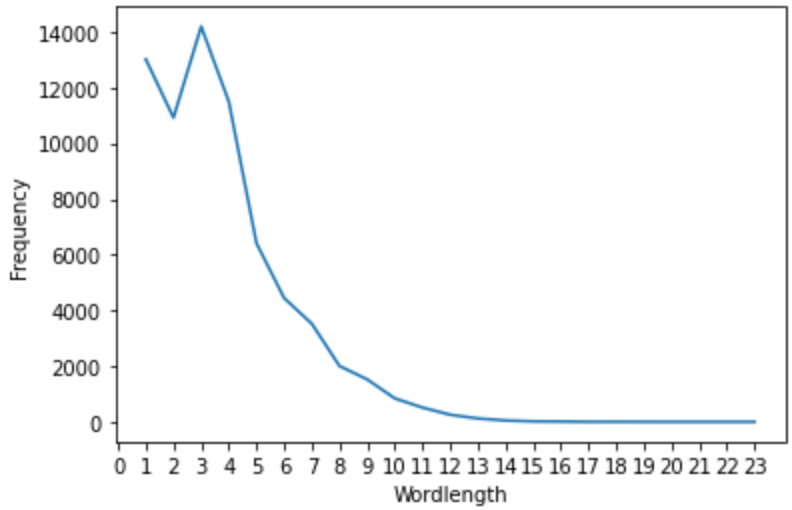
The longest word in the text T1 is a word connected by two hyphens. The word itself contains three smaller length words(self-respect-everything).

We get the frequency distribution dictionary and we can sort this dictionary by using the sorted() function available. After sorting, the frequency distribution can be represented in a plot by the following code:

```
lists_t1 = sorted(wordlen_freq_t1.items())
x1,y1=zip(*lists_t1)
plt.plot(x1,y1)
plt.xticks(range(0,24))
plt.rcParams["figure.figsize"] = (10,5)
plt.xlabel("Wordlength")
plt.ylabel("Frequency")
plt.show()
```

We have used the plt function from the matplotlib library available.

Thus for the first text, we get the following plot:



It can be seen from the above figure that the most occurring word length is 3. The second most frequent word length is 1 which mostly contains punctuation marks(,/./'/", etc). The number of words with word length 3 is 14193.

For text T2

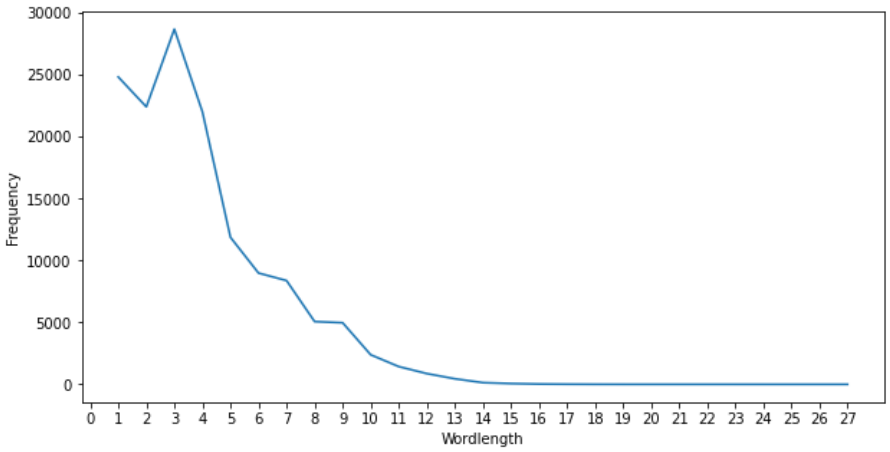
An approach similar to the one taken in making a frequency table of word length and number of words with the given length for the text T1 can be taken for text T2. The code for doing that just requires to change the variable names accordingly. The code for making the frequency table is:

```
wordlen_freq_t2 = {}
for item in tokens_t2:
  length = len(item)
  if(length == 24):
    print("24 length word : " + str(item))
  if(length == 27):
    print("27 length word : " + str(item))

  if length in wordlen_freq_t2:
    wordlen_freq_t2[length] = wordlen_freq_t2[length] + 1
  else:
    wordlen_freq_t2[length] = 1
```

```
24 length word : impartiality—deliberated
27 length word : inconveniences—cheerfulness
```

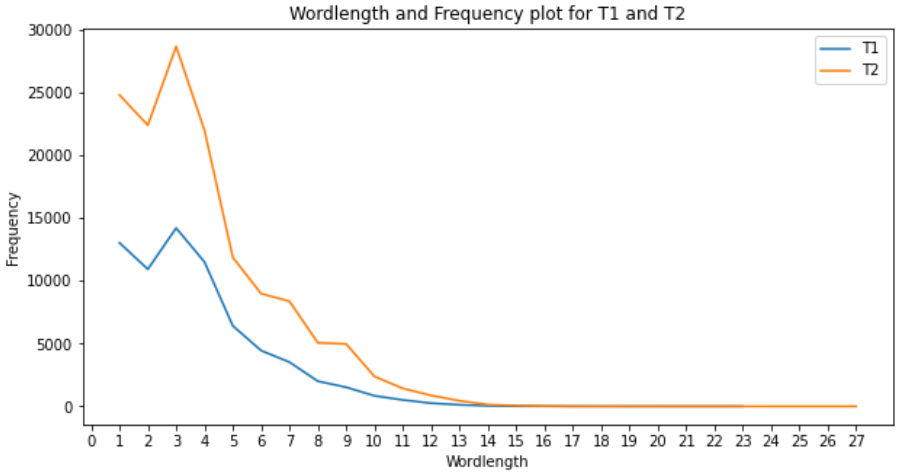
The longest word in this text is of length 27(inconveniences-cheerfulness). This also contains 2 smaller words separated by a hyphen. The plot observed in this case is :



In this text also, the most common word length is 3 with 28645 words. This text also has the second most frequent word length as 1 which mainly contains punctuation marks.

Comparison of word length/frequency distribution of both texts

The curves for both the texts can be plotted on the same plot :



As can be seen from the above plot, both the text follow the same curve pattern for word length/frequency distribution plot. The number of words with word length 9 and above is very less compared to smaller word lengths.

Project Round 2

Problem Statement:

1. Find the nouns and verbs in both the novels. Get the categories that these words fall under in the WordNet.
2. Get the frequency of each category for each noun and verb in their corresponding hierarchies and plot a histogram for the same for each novel.
3. Recognise all Persons, Location, Organisation in book. Use performance measures to measure the performance of the method used - For evaluation you take a considerable amount of random passages from the Novel, do a manual labelling and then compare your result with it.
4. Extract the relationship between the entities (mainly the characters involved in the novel).

Finding and categorizing nouns and verbs

```
from nltk import pos_tag
nltk.download('averaged_perceptron_tagger')
nltk_data1 = pos_tag(tokens_t1)
```

We first import **nltk** which contains the modules to tokenize the texts, and then we use the nltk.pos_tag() function to calculate the pos tag for each token of book 1, and then we do the same for book 2.
So ***nltk_data1*** contains pos_tags for T1 and ***nltk_data2*** contains pos_tags for T2 as shown below:

```
for i in nltk_data1[0:10]:
  print(i)
```

```
('mr.', 'NN')
('sherlock', 'NN')
('holmes', 'NNS')
(',', ',')
('who', 'WP')
('was', 'VBD')
('usually', 'RB')
('very', 'RB')
('late', 'RB')
('in', 'IN')
```

```
for i in nltk_data2[0:10]:
  print(i)
```

```
('it', 'PRP')
('is', 'VBZ')
('a', 'DT')
('truth', 'NN')
('universally', 'RB')
('acknowledged', 'VBD')
(',', ',')
('that', 'IN')
('a', 'DT')
('single', 'JJ')
```

For text T1

Now to count the tags, we use the package ***Counter*** from the ***collection*** module. A counter is a dictionary subclass which works on the principle of key-value operation and in the following words are the key and tags are the value and counter will count each tag total count present in the text.Following is the dictionary counter for T1.

```
from collections import Counter
counts = Counter(tag for word,tag in nltk_data1)
counts
```

```
Counter({'$': 1,
         "''": 55,
         '(': 5,
         ')': 5,
         ',': 3433,
         '.': 3199,
         ':': 61,
         'CC': 2242,
         'CD': 371,
         'DT': 6168,
         'EX': 242,
         'FW': 25,
         'IN': 7706,
         'JJ': 5022,
         'JJR': 144,
         'JJS': 77,
         'MD': 1176,
         'NN': 11573,
         'NNP': 1113,
         'NNPS': 2,
         'NNS': 2062,
         'PDT': 100,
         'PRP': 4422,
         'PRP$': 1804,
         'RB': 3344,
         'RBR': 98,
         'RBS': 43.
```

Now we create a list, ***list_noun1*** which is a list of all the words tagged as nouns in the T1 as shown below, there are around 14750 (NN, NNP,NNPS,NNS) tokens in T1 tagged as nouns.

```
[116] list_noun1 = []
      for item in nltk_data1:
        if(item[1]=="NN" or item[1]=="NNP" or item[1]=="NNPS" or item[1]=="NNS"):
          list_noun1.append(item)
      ##size(list_noun)
```

```
[117] len(list_noun1)

      14750
```

Now we create a dictionary ***dict_noun1***, that consists of the counts of each of the 26 categories of nouns present in text T1. The 26 categories are as follows:

noun.Tops	unique beginner for nouns
noun.act	nouns denoting acts or actions
noun.animal	nouns denoting animals
noun.artifact	nouns denoting man-made objects
noun.attribute	nouns denoting attributes of people and objects
noun.body	nouns denoting body parts
noun.cognition	nouns denoting cognitive processes and contents
noun.communication	nouns denoting communicative processes and contents
noun.event	nouns denoting natural events
noun.feeling	nouns denoting feelings and emotions
noun.food	nouns denoting foods and drinks
noun.group	nouns denoting groupings of people or objects
noun.location	nouns denoting spatial position
noun.motive	nouns denoting goals
noun.object	nouns denoting natural objects (not man-made)
noun.person	nouns denoting people
noun.phenomenon	nouns denoting natural phenomena
noun.plant	nouns denoting plants
noun.possession	nouns denoting possession and transfer of possession
noun.process	nouns denoting natural processes
noun.quantity	nouns denoting quantities and units of measure
noun.relation	nouns denoting relations between people or things or ideas
noun.shape	nouns denoting two and three dimensional shapes
noun.state	nouns denoting stable states of affairs
noun.substance	nouns denoting substances
noun.time	nouns denoting time and temporal relations

Why is wordnet.synsets() used?

Here we use ***wordnet.synsets()*** function, as here we do not perform word sense disambiguation(***WSD***), as it is a np-hard problem. So we use synset instances which are the groupings of synonymous words that express the same concept and can replace each other in a sentence. So we consider all possible tags for a particular word, without taking in consideration the problem of WSD.

```
dict_noun1 = {}
from nltk.corpus import wordnet

for item in list_noun1:
  syn = wordnet.synsets(item[0])
  for i in syn:
    if i.lexname()[0]=='n':
      if i.lexname() in dict_noun1:
        dict_noun1[i.lexname()]+=1
      else:
        dict_noun1[i.lexname()]=1

print(dict_noun1)
print(len(dict_noun1))
```

So we have the count of the 26 noun categories stored in ***dict_noun1*** as follows:

```
'noun.communication': 7382, 'noun.person': 10266, 'noun.time': 3180,
'noun.event': 1403, 'noun.cognition': 4502, 'noun.motive': 166,
'noun.state': 2676, 'noun.food': 422, 'noun.group': 2681,
'noun.artifact': 7369, 'noun.object': 1432, 'noun.substance': 1555,
'noun.quantity': 2478, 'noun.plant': 265, 'noun.body': 1255,
'noun.act': 4421, 'noun.attribute': 3280, 'noun.possession': 653,
'noun.process': 120, 'noun.animal': 953, 'noun.phenomenon': 646,
'noun.location': 2793, 'noun.relation': 396, 'noun.feeling': 650,
'noun.Tops': 434, 'noun.shape': 352}

26
```

Now we create a list, ***list_verb1*** which is a list of all the words tagged as verbs in the T1 as shown below, there are around 11797(VB, VBD, VBG, VBN, VBP, VBZ) tokens in T1 tagged as verbs.

```
[136] list_verb1 = []
      for item in nltk_data1:
        if(item[1]=="VB" or item[1]=="VBD" or item[1]=="VBG" or item[1]=="VBN" or item[1]=="VBP" or item[1]=="VBZ"):
          list_verb1.append(item)
```

```
len(list_verb1)
```

```
11797
```

As done above for the case of noun categorization, we do similar steps for verb categorization. So we again use wordnet.synsets() function because of the reason explained above.

```
dict_verb1 = {}
from nltk.corpus import wordnet

for item in list_verb1:
  syn = wordnet.synsets(item[0])
  for i in syn:
    if i.lexname()[0]=='v':
      if i.lexname() in dict_verb1:
        dict_verb1[i.lexname()]+=1
      else:
        dict_verb1[i.lexname()]=1

print(dict_verb1)
print(len(dict_verb1))
```

So we have the count of the 15 verb categories stored in ***dict_verb1*** as follows:

```
'verb.stative': 41634, 'verb.social': 11684, 'verb.possession':
15997, 'verb.change': 10516, 'verb.communication': 12698,
'verb.contact': 7290, 'verb.cognition': 9981, 'verb.competition':
2207, 'verb.creation': 6818, 'verb.consumption': 2502,
'verb.perception': 5948, 'verb.body': 6705, 'verb.motion': 8978,
'verb.emotion': 1112, 'verb.weather': 180}
15
```

For text T2

Now we repeat all the steps performed for text T1, to perform categorization of nouns and verbs. As done in T1 we count the tags, using the package ***Counter*** from the ***collection*** module, as shown below.

```
from collections import Counter
counts = Counter(tag for word,tag in nltk_data2)
counts
```

```
Counter({"''": 55,
         '(': 18,
         ')': 18,
         ',': 9131,
         '.': 5036,
         ':': 1670,
         'CC': 5070,
         'CD': 540,
         'DT': 9149,
         'EX': 281,
         'FW': 222,
         'IN': 15204,
         'JJ': 9313,
         'JJR': 378,
         'JJS': 333,
         'MD': 2867,
         'NN': 21424,
         'NNP': 2061,
         'NNS': 3650,
         'PDT': 287,
         'PRP': 9153,
         'PRP$': 4815,
         'RB': 8844,
         'RBR': 273,
         'RBS': 141,
         'RP': 302,
         'TO': 4124,
```

Now we create a list, ***list_noun2*** which is a list of all the words tagged as nouns in the T2 as shown below, there are around 27135 tokens in T2 tagged as nouns.

```
[139] list_noun2 = []
      for item in nltk_data2:
        if(item[1]=="NN" or item[1]=="NNP" or item[1]=="NNPS" or item[1]=="NNS"):
          list_noun2.append(item)

      len(list_noun2)
```

```
27135
```

Here we create dict_noun2 using wordnet.synsets() as done for T1.

```
dict_noun2 = {}
from nltk.corpus import wordnet

for item in list_noun2:
  syn = wordnet.synsets(item[0])
  for i in syn:
    if i.lexname()[0]=='n':
      if i.lexname() in dict_noun2:
        dict_noun2[i.lexname()]+=1
      else:
        dict_noun2[i.lexname()]=1

print(dict_noun2)
print(len(dict_noun2))
```

So we have the count of the 26 noun categories stored in ***dict_noun2*** as follows:

```
'noun.cognition': 9038, 'noun.state': 4757, 'noun.communication':
12290, 'noun.attribute': 7095, 'noun.person': 13034, 'noun.animal':
671, 'noun.location': 3263, 'noun.artifact': 6455, 'noun.act': 8568,
'noun.Tops': 843, 'noun.motive': 260, 'noun.phenomenon': 704,
'noun.possession': 1477, 'noun.object': 942, 'noun.feeling': 2492,
'noun.group': 5365, 'noun.plant': 803, 'noun.time': 5878,
'noun.substance': 1930, 'noun.quantity': 3215, 'noun.relation': 955,
'noun.event': 2277, 'noun.body': 869, 'noun.process': 246,
'noun.shape': 447, 'noun.food': 716}

26
```

Now we create a list, ***list_verb2*** which is a list of all the words tagged as verbs in the T2 as shown below, there are around 25549 tokens in T2 tagged as verbs.

```
list_verb2 = []
for item in nltk_data2:
  if(item[1]=="VB" or item[1]=="VBD" or item[1]=="VBG" or item[1]=="VBN" or item[1]=="VBP" or item[1]=="VBZ"):
    list_verb2.append(item)

len(list_verb2)
```

```
25549
```

As done above for the case of noun categorization, we do similar steps for verb categorization. We use ***wordnet.synsets()*** function, as we cannot successfully perform the task of word sense disambiguation because ***WSD*** is a NP-hard problem, so we take into consideration all possible word senses for a particular word.

```
dict_verb2 = {}
from nltk.corpus import wordnet

for item in list_verb2:
  syn = wordnet.synsets(item[0])
  for i in syn:
    if i.lexname()[0]=='v':
      if i.lexname() in dict_verb2:
        dict_verb2[i.lexname()]+=1
      else:
        dict_verb2[i.lexname()]=1

print(dict_verb2)
print(len(dict_verb2))
```

So we have the count of the 15 verb categories stored in ***dict_verb2*** as follows:

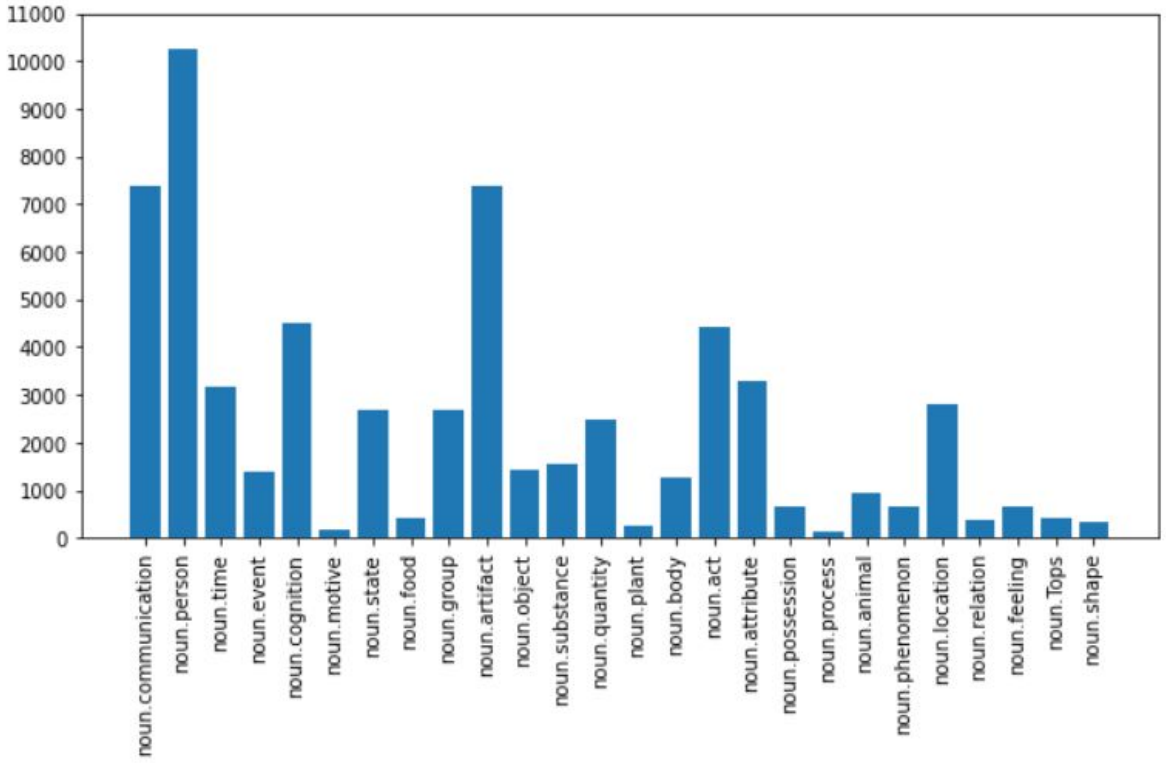
```
'verb.stative': 89811, 'verb.social': 24348, 'verb.possession':
32861, 'verb.communication': 27740, 'verb.cognition': 22592,
'verb.contact': 10965, 'verb.motion': 13700, 'verb.competition':
3166, 'verb.creation': 14545, 'verb.change': 18477, 'verb.body':
13069, 'verb.perception': 11116, 'verb.consumption': 4926,
'verb.emotion': 3184, 'verb.weather': 76}
```

Histograms for Book 1 :

For nouns :

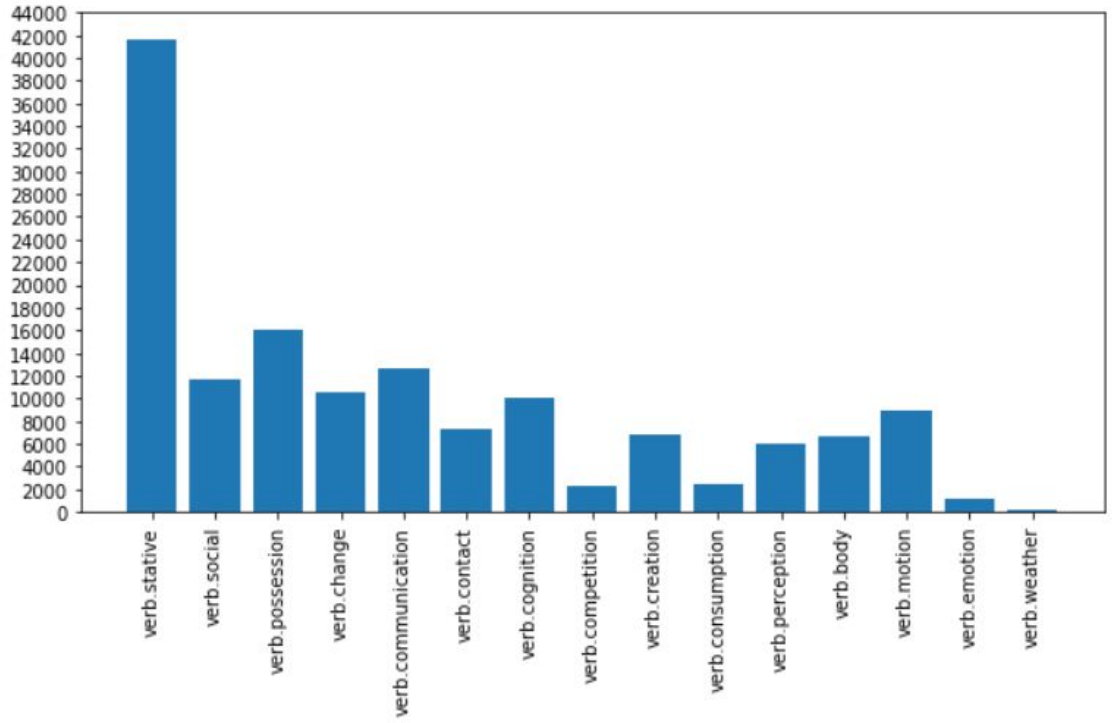
```
import matplotlib.pyplot as plt
import numpy as np

plt.bar(range(len(dict_noun1)), list(dict_noun1.values()), align = 'center')
plt.xticks(range(len(dict_noun1)), list(dict_noun1.keys()), rotation = 90)
plt.yticks(np.arange(0,12000,1000))
```



For Verbs:

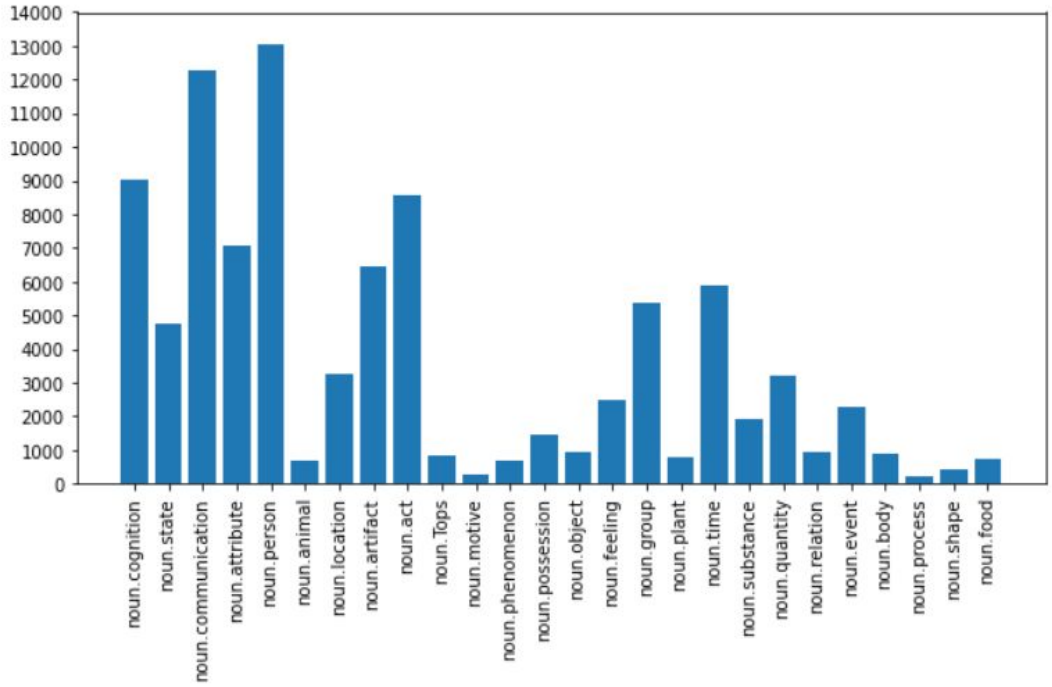
```
plt.bar(range(len(dict_verb1)), list(dict_verb1.values()), align = 'center')
plt.xticks(range(len(dict_verb1)), list(dict_verb1.keys()), rotation = 90)
plt.yticks(np.arange(0,45000,2000))
```



Histograms for Book 2:

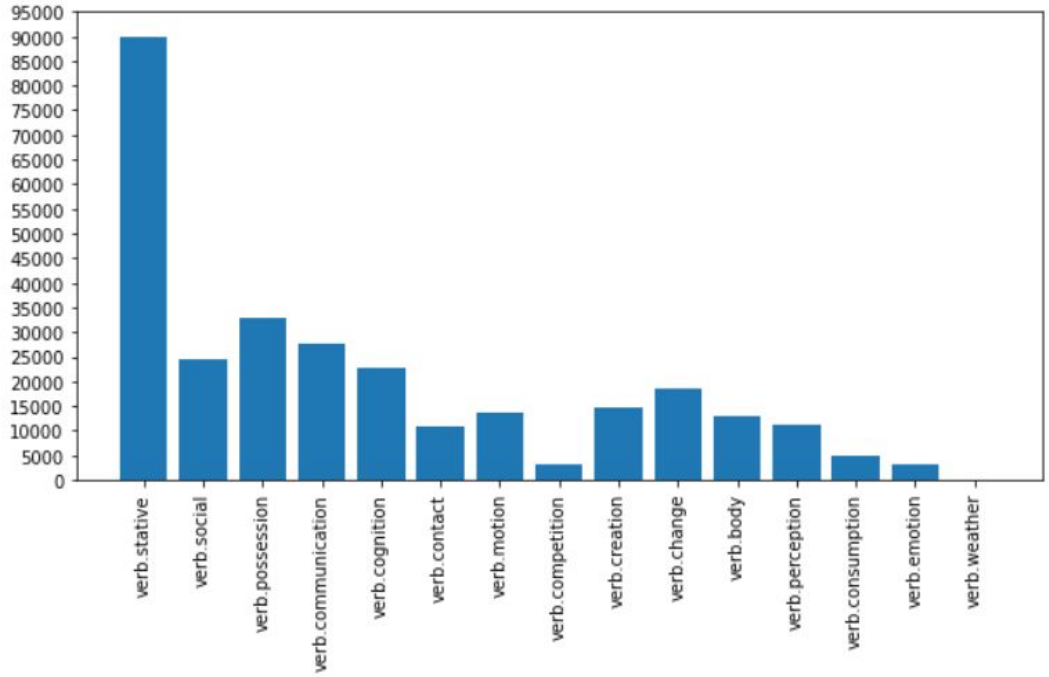
For nouns:

```
74] plt.bar(range(len(dict_noun2)), list(dict_noun2.values()), align = 'center')
plt.xticks(range(len(dict_noun2)), list(dict_noun2.keys()), rotation = 90)
plt.yticks(np.arange(0,15000,1000))
```



For Verbs:

```
plt.bar(range(len(dict_verb2)), list(dict_verb2.values()), align = 'center')
plt.xticks(range(len(dict_verb2)), list(dict_verb2.keys()), rotation = 90)
plt.yticks(np.arange(0,100000,5000))
```



From the above histograms it can be clearly seen, which category is occurring more often.

Recognizing and classifying entities

Recognising all entities like Persons, Location, Organisation in book.

```
[250] import spacy
      from spacy import displacy
      import en_core_web_sm
      nlp = en_core_web_sm.load()
```

We import ***spacy***, a free open source library for Natural Language Processing in Python. Thus we use the named entity recognition algorithm provided by spacy.

```
[ ]  doc = nlp(joined_book1)
     print([(X.text, X.label_) for X in doc.ents])
```

Following is the output we receive from above code block:

```
[('Sherlock Holmes', 'PERSON'), ('all night', 'TIME'), ('the night',
'TIME'), ('inch', 'NORP'), ('James Mortimer', 'PERSON'), ('M.R.C.S.',
'GPE'), ('C.C.H.', 'ORG'), ('1884', 'WORK_OF_ART'), ('Watson',
'PERSON'), ('Holmes', 'PERSON'), ('Watson', 'PERSON'), ('Mortimer',
'PERSON'), ('Holmes', 'PERSON'), ('Why so?', 'WORK_OF_ART'),
('Holmes', 'PERSON'), ('C.C.H.', 'ORG'), ('the Something Hunt',
'PERSON'), ('Really, Watson, you excel yourself', 'WORK_OF_ART'),
('Holmes', 'PERSON'), ('a few minutes', 'TIME'), ('one', 'CARDINAL'),
('two', 'CARDINAL'), ('Watson', 'PERSON'), ('Watson', 'PERSON'),
('C.C.', 'GPE'), ('Charing Cross', 'ORG'), ('C.C.H.', 'WORK_OF_ART'),
('Charing Cross Hospital', 'ORG'), ('Mortimer', 'PERSON'), ('_',
'CARDINAL'), ('_', 'CARDINAL'), ('London', 'GPE'), ('five years ago',
'DATE'), ('Watson', 'PERSON'), ('under thirty', 'DATE'), ('Sherlock
Holmes', 'ORG'), ('the Medical Directory', 'ORG'), ('Mortimers',
'ORG'), ('only one', 'CARDINAL'), ('Mortimer, James, M.R.C.S., 1882',
'WORK_OF_ART'), ('Dartmoor', 'GPE'), ('House', 'ORG'), ('1882 to',
'DATE'), ('1884', 'DATE'), ('Charing Cross Hospital', 'ORG'),
('Jackson', 'PERSON'), ('Comparative Pathology',
'ORG').............]
```

In the code block shown below, we have taken a random paragraph from the text where the entities are recognized and their entity tags are displayed in a highlighted manner.

```
[101] displacy.render(nlp(str(sentences[75:95])), jupyter=True, style='ent')
```

[From my small medical shelf I took down the Medical Directory ORG and turned up the name., There were several Mortimers ORG , but only one CARDINAL who could be our visitor., I read his record aloud. , " Mortimer PERSON , James PERSON , M.R.C.S. GPE , 1882 DATE , Grimpen ORG , Dartmoor GPE , Devon. House ORG -surgeon, from 1882 to DATE 1884 DATE , at Charing Cross Hospital ORG . , Winner of the Jackson prize for Comparative Pathology WORK_OF_ART , with essay entitled ' Is Disease a Reversion WORK_OF_ART ?' , Corresponding member of the Swedish Pathological Society ORG . , Author of 'Some Freaks of Atavism' (, _ Lancet ORG _ 1882 DATE). , ' Do We Progress WORK_OF_ART ?', (_ Journal PERSON of , Psychology_, March, 1883)., Medical Officer for the parishes of Grimpen, Thorsley, and High Barrow PERSON ."., "No mention of that local hunt, Watson PERSON ," said Holmes PERSON with a mischievous smile, "but a country doctor, as you very astutely observed., I think that I am fairly justified in my inferences., As to the adjectives, I said, if I remember right, amiable, unambitious, and absent-minded., It is my experience that it is only an amiable man in this world who receives testimonials, only an unambitious one who abandons a London GPE career for the country, and only an absent-minded one who leaves his stick and not his visiting-card after waiting an hour TIME in your room.", "And the dog?", "Has been in the habit of carrying this stick behind his master., Being a heavy stick the dog has held it tightly by the middle, and the marks of his teeth are very plainly visible., The dog's jaw, as shown in the space between these marks, is too broad in my opinion for a terrier and not broad enough for a mastiff.]

Measuring accuracy of the Spacy NER algorithm

For measuring the accuracy, we choose a paragraph at random, and recognize the entities in that paragraph and manually label them. We then run the spacy's entity recognition algorithm on the same paragraph we have chosen. Now we check for the entities that have been recognized differently by spacy's algorithm as compared to our manually labelled list of entities.

```
[96] doc1 = nlp(parra_book1)
     print([(X.text, X.label_) for X in doc1.ents])
```

From the above code block, we get the following list of entities in the paragraph that have been identified by the spacy's algorithm. 24 entities have been recognised by the algorithm in the chosen paragraph.

```
[('Sherlock Holmes', 'PERSON'), ('all night', 'TIME'), ('the night',
'TIME'), ('inch', 'NORP'), ('James Mortimer', 'PERSON'), ('M.R.C.S.',
'GPE'), ('C.C.H.', 'ORG'), ('1884', 'WORK_OF_ART'), ('Watson',
'PERSON'), ('Holmes', 'PERSON'), ('Watson', 'PERSON'), ('Mortimer',
'PERSON'), ('Holmes', 'PERSON'), ('Why so?', 'WORK_OF_ART'),
('Holmes', 'PERSON'), ('C.C.H.', 'ORG'), ('the Something Hunt',
'PERSON'), ('Really, Watson, you excel yourself', 'WORK_OF_ART'),
('Holmes', 'PERSON'), ('a few minutes', 'TIME'), ('one', 'CARDINAL'),
('two', 'CARDINAL'), ('Watson', 'PERSON'), ('Watson', 'PERSON')]
```

Following is the list in which we have correctly labelled the entities which have been incorrectly identified by the spacy's algorithm.

```
manual_tagged_list =
[('inch','QUANTITY'),('M.R.C.S.','PRODUCT'),('1884','DATE'),('Why
so?','NULL'),('the Something Hunt','EVENT'),('the local
hunt','EVENT'),('Watson','PERSON')]
```

So as we can see that out of the 24 entities recognized by the algorithm, 7 have been incorrectly identified as can be seen in ***manual_tagged_list***.

```
correctly_labelled = len(doc1.ents)-len(manual_tagged_list)
Accuracy = correctly_labelled / len(doc1.ents)
print(Accuracy)
```

```
0.7083333333333334
```

So in the above code block we have calculated the ***Accuracy*** of our algorithm, where

- ***len(doc1.ents)***: length of the list of entities recognized by our algorithm in the paragraph.
- ***len(manual_tagged_list)***: length of the list of entities incorrectly identified by the spacy's algorithm.

As can be seen above, we have an ***Accuracy*** around ***70.83%***, which is a pretty good accuracy for a named entity recognition algorithm.

Finding Relations between the Entities.

For this task, we use stanza which allows users to access the Java toolkit developed for NLP by Stanford University. Stanza allows to create a server as a background process for python code which then sends requests to the Stanford CoreNLP toolkit which can be used to determine relations between entities annotations can be obtained by the users by writing python code without worrying about the Java implementation.

CoreNLP is implementation of NLP tasks in Java including linguistic annotations for text, including token and sentence boundaries, parts of speech, named entities, numeric and time values etc. it follows a pipeline.

For our code, we first import Stanza and from that we import CoreNLPClient which allows us to make a connection to the server which has the Java toolkit support needed to perform the required NLP tasks. In initializing the client as shown in the code block below, we keep the timeout value large as a large text part is to be annotated. The annotator used is “openie” which supports finding relationships between entities. The annotated text output is converted into json format and stored in the variable named *document*.

```
import stanza
from stanza.server import CoreNLPClient
stanza.install_corenlp()
client = CoreNLPClient(timeout=150000000, be_quiet=True, annotators=['openie'], endpoint='http://localhost:9003')
document = client.annotate(joined_book1[0:50000], output_format='json')
triples = []
import en_core_web_sm
nlp = en_core_web_sm.load()
for sentence in document['sentences']:
    for triple in sentence['openie']:
      doc=nlp(triple['subject'])
      doc1=nlp(triple['object'])
      y=([X.label_ for X in doc.ents ])
      y1=([X.label_ for X in doc1.ents])
      #print(y,y1)
      if (y and y1 ):
        triples.append({
            'subject': (y,triple['subject']),
             'relation': triple['relation'],
              'object': (y1,triple['object'])
         })

for i in triples:
  print(i)
  print()
```

The list *triples* contains all the relationships recognized by the annotator. It contains the relationships in the format: **(subject: Entity1, Relation,object: Entity2)** . To find the relations, we have only included output from the first 50000 characters input taken in the annotator to reduce the calculation time, otherwise the server would have timed out. So we have to break the input into multiple parts for each book to annotate the whole book.

For the first book, the output of annotations is seen as follows on the next page:


```
{'subject': (['ORDINAL'], 'that second highest expert'), 'relation': 'is in', 'object': (['LOC'], 'Europe')}
{'subject': (['CARDINAL'], 'one Michaelmas'), 'relation': 'is with', 'object': (['CARDINAL'], 'five of his idle companions')}
{'subject': (['EVENT'], 'Whereat Hugo'), 'relation': 'ran from', 'object': (['ORG'], 'house')}
{'subject': (['EVENT'], 'Whereat Hugo'), 'relation': 'giving', 'object': (['ORG'], 'kerchief of maid ’s')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'had resided at', 'object': (['ORG'], 'Baskerville Hall')}
{'subject': (['PERSON'], 'Sir Charles Baskerville'), 'relation': 'was in', 'object': (['PERSON'], 'habit night')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'had declared On', 'object': (['ORDINAL'], 'fourth of May')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'had declared', 'object': (['DATE', 'GPE'], 'his intention of starting next day for London')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'had declared On', 'object': (['ORDINAL'], 'fourth')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'had declared', 'object': (['GPE'], 'his intention for London')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'had declared', 'object': (['DATE'], 'his intention of starting next day')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'had declared', 'object': (['GPE'], 'his intention of starting day for London')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'had declared', 'object': (['DATE', 'GPE'], 'his intention of next day for London')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'had declared', 'object': (['DATE'], 'his intention of next day')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'had declared', 'object': (['DATE', 'GPE'], 'his intention of day for London')}
{'subject': (['PERSON'], 'Sir Charles ’s footmarks'), 'relation': 'were traced down', 'object': (['ORG'], 'alley')}
{'subject': (['PERSON'], 'Sir Charles ’s footmarks'), 'relation': 'were easily traced down', 'object': (['ORG'], 'alley')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'was about go to', 'object': (['GPE'], 'London')}
{'subject': (['PERSON'], 'Sir Charles'), 'relation': 'was go to', 'object': (['GPE'], 'London')}
{'subject': (['CARDINAL'], 'one false statement'), 'relation': 'was', 'object': (['GPE'], 'made by Barrymore at inquest')}
{'subject': (['PERSON'], 'yew hedge'), 'relation': 'is', 'object': (['CARDINAL'], 'penetrated at one point by gate')}
{'subject': (['PERSON'], 'Rodger Baskerville'), 'relation': 'youngest of', 'object': (['CARDINAL'], 'three brothers')}
{'subject': (['PERSON'], 'Dr. Mortimer'), 'relation': 'will', 'object': (['DATE'], 'tomorrow will obliged to you')}
{'subject': (['PERSON'], 'Dr. Mortimer'), 'relation': 'will', 'object': (['DATE'], 'tomorrow will obliged')}
{'subject': (['PERSON'], 'Dr. Mortimer'), 'relation': 'will', 'object': (['DATE'], 'tomorrow will much obliged')}
{'subject': (['PERSON'], 'Dr. Mortimer'), 'relation': 'will', 'object': (['DATE'], 'tomorrow will much obliged to you')}
```

For the second book, the output of the entities is as follows on the next page:


```
{'subject': (['PERSON'], 'My dear Mr. Bennet'), 'relation': 'said at_time', 'object': (['DATE'], 'one day')}
{'subject': (['PERSON'], 'My Mr. Bennet'), 'relation': 'said at_time', 'object': (['DATE'], 'one day')}
{'subject': (['PERSON'], 'Mrs. Bennet'), 'relation': 'is with', 'object': (['CARDINAL'], 'assistance of her five daughters')}
{'subject': (['PERSON'], 'Mr. Bingley'), 'relation': 'returned In', 'object': (['DATE'], 'few days')}
{'subject': (['PERSON'], 'Mr. Bingley'), 'relation': 'sat', 'object': (['TIME'], 'about ten minutes with him')}
{'subject': (['PERSON'], 'Mr. Bingley'), 'relation': 'returned', 'object': (['PERSON'], 'Mr. Bennet ’s visit')}
{'subject': (['PERSON'], 'Mr. Bingley'), 'relation': 'sat', 'object': (['TIME'], 'about ten minutes')}
{'subject': (['PERSON'], 'Mr. Bingley'), 'relation': 'was obliged', 'object': (['DATE'], 'to town day consequently')}
{'subject': (['PERSON'], 'Mr. Bingley'), 'relation': 'was obliged', 'object': (['DATE'], 'to town day unable')}
{'subject': (['PERSON'], 'Mr. Bingley'), 'relation': 'bring', 'object': (['CARDINAL'], 'twelve ladies')}
{'subject': (['PERSON'], 'Mr. Darcy'), 'relation': 'danced once with', 'object': (['PERSON'], 'Mrs. Hurst')}
{'subject': (['PERSON'], 'Mr. Darcy'), 'relation': 'danced with', 'object': (['PERSON'], 'Mrs. Hurst')}
{'subject': (['PERSON'], 'Mr. Darcy'), 'relation': 'spent', 'object': (['TIME'], 'rest of evening')}
{'subject': (['PERSON'], 'Mr. Darcy'), 'relation': 'spent rest in', 'object': (['CARDINAL'], 'walking speaking occasionally to one of his party')}
{'subject': (['PERSON'], 'Mr. Darcy'), 'relation': 'declined with', 'object': (['PERSON'], 'Miss Bingley')}
{'subject': (['PERSON'], 'Mr. Darcy'), 'relation': 'danced only once with', 'object': (['PERSON'], 'Mrs. Hurst')}
{'subject': (['PERSON'], 'Mr. Darcy'), 'relation': 'declined with', 'object': (['PERSON'], 'once Miss Bingley')}
{'subject': (['PERSON'], 'Elizabeth Bennet'), 'relation': 'sit down for', 'object': (['CARDINAL'], 'two dances')}
{'subject': (['PERSON'], 'Mr. Darcy'), 'relation': 'looking at', 'object': (['PERSON'], 'eldest Miss Bennet')}
{'subject': (['PERSON'], 'Mr. Darcy'), 'relation': 'looking at', 'object': (['PERSON'], 'Miss Bennet')}
{'subject': (['ORG'], 'Elizabeth'), 'relation': 'felt', 'object': (['PERSON'], 'Jane ’s pleasure')}
{'subject': (['PERSON'], 'Mr. Bingley'), 'relation': 'inherited property to', 'object': (['QUANTITY'], 'amount of nearly hundred thousand pounds')}
{'subject': (['ORG'], 'Bingley'), 'relation': 'had reliance On', 'object': (['PERSON'], 'strength of Darcy ’s regard')}
{'subject': (['PERSON'], 'Sir William Lucas'), 'relation': 'had in', 'object': (['GPE'], 'had formerly trade in Meryton')}
{'subject': (['PERSON'], 'Sir William Lucas'), 'relation': 'had in', 'object': (['GPE'], 'had trade in Meryton')}
{'subject': (['PERSON'], 'Mrs. Bennet'), 'relation': 'is with', 'object': (['PERSON'], 'civil self command to Miss Lucas')}
{'subject': (['PERSON'], 'Miss Bennet ’s manners'), 'relation': 'grew on', 'object': (['PERSON'], 'goodwill of Mrs. Hurst')}
{'subject': (['PERSON'], 'Miss Bennet ’s pleasing manners'), 'relation': 'grew on', 'object': (['PERSON'], 'goodwill of Mrs. Hurst')}
{'subject': (['ORG'], 'Elizabeth'), 'relation': 'did', 'object': (['PERSON'], 'Sir William')}
```


As can be seen from the output, many of the relationships are identified correctly in both the books. But there are many relationships which are less accurate like:

```
{'subject': (['ORG'], 'Elizabeth'), 'relation': 'did', 'object':
(['PERSON'], 'Sir William')}
```

In the relationship seen above, Elizabeth which is clearly a person is identified as an Organization. Also “did” cannot be identified as a valid relationship.

Thus it can be seen that many relationships can be wrongly identified by the annotator. All the identified relations can be found in the code which is uploaded to github, link to which is attached with the report.