

# **NLP PROJECT**

**Team: Hats**

**Team Members:**

**Harsh Kumar: 21ucs086**

**Om Gupta: 21ucs140**

**Pitani Nitin Srichakri: 21ucs149**

**Tarun Agarwal: 21ucs219**

**Book: “The White Company”- Arthur Conan Doyle**

**Submitted to: Dr. Shakti Balan**

**ACKNOWLEDGEMENT**

We are grateful to our respected teacher, Dr. Sakthi Balan, whose provided knowledge in classes benefited us in completing this project successfully. Thank you so much for your continuous support and presence whenever needed. Lastly, we would like to thank everyone involved in the project directly or indirectly. And we are always thankful to The Great Almighty for always having a blessing on us

## AIM:

This project aims to obtain PoS tagging on a Novel, calculate bigram probability, and play the Shannon game.

## Data Description:

The novel we use for this project is “The white company”- Arthur Canon Doyle. Here is complete [link](#) to the project.

```
# Read book from a file
t1 = open("/book1.txt", encoding="utf8")
```

Firstly, importing some libraries for various tasks

- Importing the ‘pandas’ module for data analysis and machine learning tasks.
- Importing ‘nltk’(natural language toolkit) for the NLP-related tasks.
- Importing ‘matplotlib.pyplot’ and “Seaborn” for graphs and frequency analysis.

```
!pip install pattern
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
```

```
import operator
import nltk
from matplotlib import pyplot as plt
import string
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
from collections import Counter, defaultdict
import seaborn as sns
import pandas as pd
import numpy as np
from wordcloud import WordCloud, STOPWORDS
from pattern.text.en import singularize
import inflect
```

## Initialization and Processing:

This code initializes empty lists ('text', 'words', 'final', 'nouns', 'verbs') for storing text data and linguistic elements, along with objects like a lemmatizer ('lemmatizer') and an inflect engine ('p').

- Tokenization and Preprocessing: It processes a list of sentences in the 'text' variable.
- Cleaning: Removes punctuation and digits from the sentence.
- Tokenization: Breaks the cleaned sentence into individual words.
- Part of Speech Tagging: Tags each word with its part of speech.
- Filtering and Preprocessing:
  - Filters and preprocesses words by singularizing them, removing stopwords, ignoring short words, and checking against a custom list.
  - The filtered words and their part-of-speech tags are added to the 'words' list.

```
# Initialize lists and objects
text = []
words = []
final = []
nouns = []
verbs = []
lemmatizer = WordNetLemmatizer()
custom = ["chapter", "page"]
p = inflect.engine()
```

```
# Process and tokenize sentences
for sentence in text:
    # Remove punctuation and digits
    punctuationfree = "".join([i for i in sentence if i not in string.punctuation and not i.isdigit()])
    # Tokenize words
    word_tokens = word_tokenize(punctuationfree)
    # Perform part of speech tagging
    filtered_sentence = nltk.pos_tag(word_tokens)
    # Filter and preprocess words, removing stopwords and applying singularization
    filtered_sentence = [(singularize(w.casefold()), t) for w, t in filtered_sentence if not w.lower() in STOPWORDS and len(w) > 1 and w.lower() not in custom]
    words.append(filtered_sentence)
```

## Lemmatization:

This code lemmatizes words in a list of sentences based on their part-of-speech (POS) tags and stores the lemmatized words and their original POS tags in the 'final' list.

```
# Lemmatize words based on their POS tags
for sentence in words:
    temp = []
    for word, tag in sentence:
        temp.append((lemmatizer.lemmatize(word, get_wordnet_pos(tag)), tag))
    final.append(temp)
```

## Plotting Frequencies:

- This plots the word frequency of the words in a list of preprocessed sentences ('words').
  - It creates a bar plot showing the top 25 most frequently occurring words, with words on the y-axis and their frequencies on the x-axis.
  - The code uses Python libraries like 'Counter', 'pandas', and 'seaborn' for data manipulation and visualization.

```
# Plotting Word frequency
to_plot = []
for sentence in words:
    for word, tag in sentence:
        to_plot.append(word)

counted = Counter(to_plot)
word_freq = pd.DataFrame(counted.items(), columns=['word', 'frequency']).sort_values(by='frequency', ascending=False)

word_freq = word_freq.head(25)
plt.title("Word frequency")
sns.barplot(x='frequency', y='word', data=word_freq)
plt.show()
```

- Combining words from the 'final' list into a single string called 'word\_text' by joining them with spaces creates a continuous text.

```
# Combine the words in 'final' into a single string
word_text = ' '.join([word for sentence in final for word, _ in sentence])
```

- Creating a WordCloud visualization from the 'word\_text' string, where words are represented graphically, without using any custom stopwords.
  - The WordCloud has specific dimensions (800x400 pixels) and a white background.

```
# Create a WordCloud object without custom stopwords
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(word_text)
```

- Displaying a Word Cloud visualization using Matplotlib.
  - 0 It sets the size of the figure, shows the Word Cloud image with a specified interpolation method, removes the axis, sets a title for the plot, and finally displays the Word Cloud.

```
# Display the word cloud using matplotlib
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("Word Cloud")
plt.show()
```

- Plotting the frequency of Treebank tags for words in the 'words' dataset:
  - It collects Treebank tags for each word in the sentences.
  - 0 Counts the frequency of each tag and creates a sorted DataFrame.
  - Displays the top 10 most frequent tags in a bar plot using Seaborn.

```
# Plotting tag frequency (Treebank Tags)
to_plot = []
to_plot_tag = []
for sentence in words:
    for word, tag in sentence:
        to_plot_tag.append(tag)

counted_tag = Counter(to_plot_tag)
tag_freq = pd.DataFrame(counted_tag.items(), columns=['tag', 'frequency']).sort_values(by='frequency', ascending=False)

tag_freq = tag_freq.head(10)
plt.title("Tag Frequency (Treebank Tags)")
sns.barplot(x='frequency', y='tag', data=tag_freq)
plt.show()
```

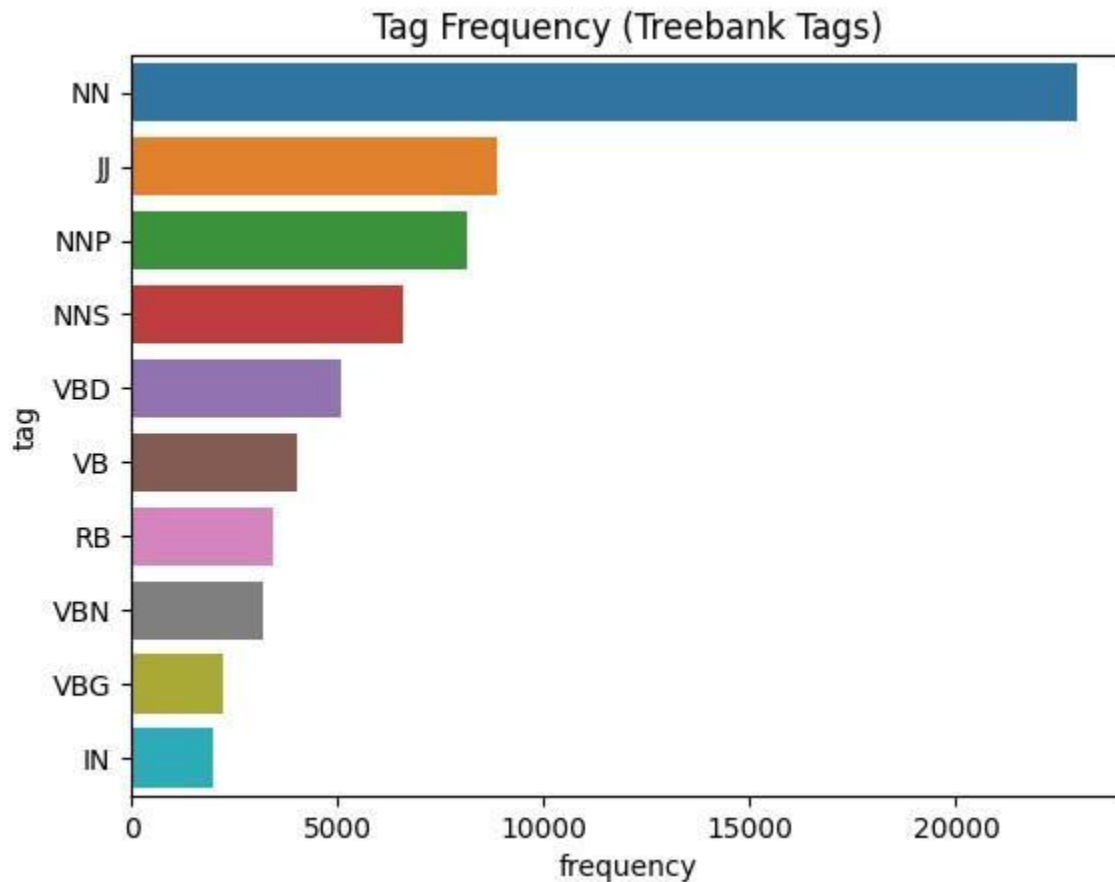
## Displaying frequencies: •

Displaying the word frequency:





- **Tag frequency:**



## Finding the Largest Chapter in the Book:

- The code begins by importing necessary libraries: 're' for regular expressions and 'defaultdict' for a dictionary with default values.
- It opens and reads the content of a text file located at '/book1.txt' into the 'data' variable.
- Splitting Chapters:
  - The code assumes that chapters in the book are separated by the text "Chapter {number}".



- 0 It uses a regular expression ('re.split()') to split the text into chapters. The '[39:]' slice is used to exclude content before the 40th occurrence of "Chapter," which is used to skip any table of contents or preambles.
- Storing Chapter Lengths:
  - 0 It initializes a dictionary called 'chapter\_lengths' to store the length (word count) of each chapter.
    - A global variable 'corpus' is initialized with the 10th chapter's content.
- The code iterates through the chapters, counting the words in each chapter and storing the counts in the 'chapter\_lengths' dictionary.
- It identifies the chapter with the maximum word count using the 'max()' function and stores the chapter number and word count in 'largest\_chapter' and 'length', respectively.
- The function returns the chapter number and word count of the largest chapter.

```

import re
from collections import defaultdict

def get_largest_chapter(file_path):
    with open('/book1.txt', 'r') as file:
        data = file.read()

    # Assuming chapters are separated by 'Chapter {number}'
    chapters = re.split(r'Chapter', data)[39:]
    # Store chapter lengths
    chapter_lengths = defaultdict(int)
    global corpus
    corpus = chapters[9]
    for i, chapter in enumerate(chapters):
        # Remove whitespace and count words
        words = re.findall(r'\b\w+\b', chapter)
        chapter_lengths[i+1] = len(words)

    # Get chapter with maximum length
    largest_chapter = max(chapter_lengths, key=chapter_lengths.get)

    return largest_chapter, chapter_lengths[largest_chapter]

# Usage
file_path = '/book1.txt'
largest_chapter, length = get_largest_chapter(file_path)
print(f'The largest chapter is Chapter {largest_chapter} with {length} words.')

```

The largest chapter is Chapter 10 with 7481 words.

## Text Preprocessing Function:

This function, 'preprocesscorpus(corpus)', performs several text preprocessing steps on an input text corpus:

- Converts text to lowercase.
- Adds "eos" as a sentence boundary marker at the beginning.
- Replace periods, exclamation marks, and question marks with "eos" to mark the end of sentences.

- Removes commas, quotation marks, semicolons, hyphens, and digits.
- Removes the English possessive form "'s".

```
def preprocesscorpus( corpus):
    corpus = corpus.lower()
    corpus = "eos " + corpus
    corpus = corpus.replace(".", " eos" )
    corpus = corpus.replace("!", " eos" )
    corpus = corpus.replace("?", " eos" )
    corpus = corpus.replace(",", "")
    corpus = corpus.replace('"', "")
    corpus = corpus.replace("'", "")
    corpus = corpus.replace(";", "")
    corpus = corpus.replace("-", " ")
    corpus = corpus.replace("0", "")
    corpus = corpus.replace("1", "")
    corpus = corpus.replace("2", "")
    corpus = corpus.replace("3", "")
    corpus = corpus.replace("4", "")
    corpus = corpus.replace("5", "")
    corpus = corpus.replace("6", "")
    corpus = corpus.replace("7", "")
    corpus = corpus.replace("8", "")
    corpus = corpus.replace("9", "")
    corpus = corpus.replace("'s", "")
    return corpus

corpus = preprocesscorpus(corpus)
print(corpus)
```

## Text Tokenization and Vocabulary Generation:

1. Tokenization: It uses the NLTK library's 'word\_tokenize' function to split the input 'corpus' into individual words or tokens.

2. Vocabulary Generation: It creates a unique and sorted list of tokens, forming the vocabulary of the text. Duplicate tokens are removed, and the list is sorted.
3. Finally, it prints the distinct tokens, which represent the unique words in the text corpus.

```
from nltk import word_tokenize
def generate_token(corpus):
    tokens = word_tokenize(corpus)
    return tokens
tokens = generate_token(corpus)
dis_token = list(set(sorted(tokens)))
print(dis_token)
```

## Token Frequency Analysis:

- Frequency Count: It calculates the frequency of each token in the 'tokens' list and stores the results in a dictionary called 'dic'.
- Printing Token Frequencies: It then iterates through the dictionary and prints each token along with its frequency count.

```
def freq(tokens):
    dic={}
    for tok in tokens:
        dic[tok]=0
    for tok in tokens:
        dic[tok]+=1
    return dic
dic = freq(tokens)
for i in dic.items():
    print(i[0],"\t:",i[1])
```

## Generating Bigrams:

- It takes a list of tokens ('tokens') and an integer 'k' as input and generates bigrams, which are pairs of consecutive tokens. It stores these bigrams in a list 'l'.
- It iterates through the tokens, taking 'k' tokens at a time, and appends them to the 'l' list.
- It removes the last element from the 'l' list because the loop overshoots by one iteration.
- Finally, it prints the generated bigrams.

In summary, this code generates bigrams from a list of tokens and displays the resulting pairs of consecutive words.

```
def gen_bigram(tokens,k):  
    l=[]  
    i=0  
    while(i<len(tokens)):  
        l.append(tokens[i:i+k])  
        i=i+1  
    l=l[:-1]  
    return l  
bigram = gen_bigram(tokens,2)  
for i in bigram:  
    print(i)
```

## Bigram Frequency Analysis:

- It calculates the frequency of each unique bigram in the 'bigram' list and stores the results in a dictionary named 'dct1'.
- It then iterates through the dictionary and prints each bigram along with its frequency count.

```
def gen_bigram_freq(bigram):
    dct1={}
    for i in bigram:
        st=" ".join(i)
        dct1[st]=0
    for i in bigram:
        st=" ".join(i)
        dct1[st]+=1
    return dct1
dct1=gen_bigram_freq(bigram)
for i in dct1.items():
    print(i[0], ":", i[1])
```

## Probability Table Generation:

- It calculates the conditional probabilities of word pairs based on the provided data. The resulting probabilities are stored in a two-dimensional list 'l'.
- It uses nested loops to iterate through all distinct tokens in the 'dis\_token' list and calculate the conditional probabilities.
- For each pair of distinct tokens, it calculates the numerator by checking the frequency of the bigram in 'dct1' and the denominator by checking the frequency of the first token in 'dic'.
- The calculated probabilities are rounded to three decimal places.
- Finally, it prints the generated probability table.
- Note: The commented part prints probability table.



```

✓ [17] def find1(s,dct1):
4s      try:
          return dct1[s]
      except:
          return 0
def print_probability_table(distinct_tokens,dct,dct1):
    n=len(distinct_tokens)
    l=[[0]*n for i in range(n)]
    for i in range(n):
        denominator = dct[distinct_tokens[i]]
        for j in range(n):
            numerator = find1(distinct_tokens[i]+" "+distinct_tokens[j],dct1)
            l[i].append(float("{:.3f}".format(numerator/denominator)))
    return l

print("Number of tokens = \n")
probability_table=print_probability_table(dis_token,dic,dct1)
n=len(dis_token)
print(n)
# print("\t", end="")
# for i in range(n):
#     print(dis_token[i],end="\t")
# print("\n")
# for i in range(n):
#     print(dis_token[i],end="\t")
#     for j in range(n):
#         print(probability_table[i][j],end="\t")
#     print("\n")

Number of tokens =

2048

```

## Bigram Word Predictor:

- It takes as input:
  - 'sentence': The input sentence for which you want to predict the next word.
  - 'probability\_table': A table of conditional probabilities between pairs of words.
  - 'dis\_token': A list of distinct tokens (words).



- It checks if there's a non-empty input sentence ('if sentence:').
- If the last word of the input sentence is in the list of distinct tokens ('dis\_token'), it calculates the most likely next word based on the bigram probabilities in the 'probability\_table'.
- It returns the most likely word as the prediction or 'None' if no prediction can be made.
- The code then takes user input for a sentence and uses the 'generate\_next\_word' function to predict the next word. If a prediction is made, it's printed; otherwise, it indicates that no probable next word was found.
- The code calculates conditional probabilities based on the bigram model. It assumes that the most probable next word depends only on the previous word (bigram), which might not be accurate for all types of text.

```

✓ 9s [19] def generate_next_word(sentence, probability_table, dis_token):
    if sentence:
        last_word = sentence.split()[-1]
        if last_word in dis_token:
            last_word_index = dis_token.index(last_word)
            probabilities = probability_table[last_word_index]
            if probabilities:
                most_likely_word_index = probabilities.index(max(probabilities))
                most_likely_word = dis_token[most_likely_word_index]
                return most_likely_word
        return None

    # Example usage:
    sentence = input("Enter a sentence: ")
    next_word = generate_next_word(sentence, probability_table, dis_token)
    if next_word:
        print(f"The next most probable word after '{sentence}' is: {next_word}")
    else:
        print("No probable next word found.")

```

- It assumes that the most probable next word depends only on the previous word (bigram), which might not be accurate for all types

of text. Hence the phrases "killer is a," "victim is a," and "a" all give the same output.

```
Enter a sentence: killer is a
The next most probable word after 'killer is a' is: man
```

```
Enter a sentence: victim is a
The next most probable word after 'victim is a' is: man
```

```
Enter a sentence: a
The next most probable word after 'a' is: man
```

## Shannon Game Word Generation:

- The 'play\_shannon\_game' function takes three input parameters:
  - the 'original\_sentence,' which serves as the starting point for word generation,
  - a 'probability\_table' containing conditional probabilities between word pairs (based on a bigram model),
  - and 'dis\_token,' a list of distinct words used for word prediction.
- It initializes an empty list called 'generated\_sentence' and splits the original sentence into individual words. Then, it iterates through these words, appending each one to 'generated\_sentence' and using the 'generate\_next\_word' function to predict the next word based on the words accumulated so far.
- After generating words for all input words, the code joins them to form a new sentence, which is returned as the function's output.

```
def play_shannon_game(original_sentence, probability_table, dis_token):
    words = original_sentence.split()
    generated_sentence = []

    for word in words:
        generated_sentence.append(word)
        next_word = generate_next_word(" ".join(generated_sentence), probability_table, dis_token)
        if next_word:
            generated_sentence.append(next_word)

    generated_sentence = " ".join(generated_sentence)
    return generated_sentence

# Example usage:
original_sentence = "which seemed to proclaim that the whole army of the prince was about to emerge from the mountain passes"
generated_sentence = play_shannon_game(original_sentence, probability_table, dis_token)

print("Original Sentence:", original_sentence)
print("Generated Sentence:", generated_sentence)
```

```
Original Sentence: which seemed to proclaim that the whole army of the prince was about to emerge from the mountain passes
Generated Sentence: which the seemed to to the proclaim that i the other whole army of the the other prince was a about us to the emerge from his the other mountain passes
```

Bigram models are limited because they only consider the probability of a word based on the immediately preceding word. Even though two consecutive words make sense, the sentence as a whole does not make sense.

## Round 2

### Aim:

Recognize all the entities in a passage and then calculate the F-score by comparing it to manually labelled data.

### Libraries Used:

SpaCy is a free, open-source library for Natural Language Processing (NLP) in Python. It's used to build applications that process and understand large amounts of text.

```
import spacy
from spacy import displacy
from spacy import tokenizer
from spacy.scorer import Scorer
from spacy.tokens import Doc
from spacy.training import Example
nlp = spacy.load('en_core_web_sm')
```

## 1. Recognizing all the entities and then Calculating the F-score of same

### 1. Function 'evaluate'

This function is responsible for evaluating the performance of the NLP model based on the provided examples using spaCy's 'Scorer' and 'Example' classes.

- 'Scorer()' initializes a scorer object to evaluate the model's performance.
- The function takes a list of examples, where each example consists of a tuple containing the input text and its annotations (entities).

- It iterates through the provided examples, converts the input text into a 'Doc' object using 'nlp.make\_doc', creates a spaCy 'Example' object from this text and annotations, and appends it to the 'example' list.
- After processing all examples, it evaluates the model using 'nlp.evaluate' with the prepared list of 'Example' objects and returns the scores.

## 2. Function 'fun'

This function process text from files and perform entity extraction and evaluation.

- It takes three parameters: 'tn' (file object), 'sm' (an empty list to store extracted entities), and 'smm' (a list of example data).
- It iterates through lines in the given file ('tn'), extracts entities using the loaded spaCy model ('nlp'), and appends the extracted entities to the 'sm' list.
- Then, it calls the 'evaluate' function with the provided examples ('smm'), which seems to contain pre-defined annotated examples.
- Finally, it prints out the extracted entities ('sm') and the precision, recall, and F-score calculated from the evaluation results.

```

nlp = spacy.load('en_core_web_sm')
def evaluate(examples):
    scorer = Scorer()
    example = []

    for input_, annot in examples:
        pred = nlp.make_doc(input_)
        # print(pred)
        temp = Example.from_dict(pred, annot)
        # temp.predicted = nlp(str(example.predicted))
        example.append(temp)
    scores = nlp.evaluate(example)
    return scores

def fun(tn, sm, smm):
    text = ""
    for line in tn:
        text = line
        doc = nlp(text)
        # ents = [(e.text, e.start_char, e.end_char, e.label_) for e in doc.ents]
        temp = (line[:len(line) - 1], {'entities': [(e.start_char, e.end_char, e.label_) for e in doc.ents]}) # extracting entites from the line
        sm.append(temp)
    results = evaluate(smm) # evaluating the model
    print(sm)
    print("Precision {:.4f}\tRecall {:.4f}\tF-score {:.4f}".format(results['ents_p'], results['ents_r'], results['ents_f']))

t2 = open("passage.txt", encoding="utf8")
t3=open("passage2.txt",encoding="utf8")
t4=open("passage3.txt",encoding="utf8")

ex = []
ey=[]
ez=[]
exx = [
    ('At this order a lay-brother swung open the door, and two other lay-brothers entered leading ', {'entities': []}),
    ('headed, with a peculiar half-humorous, half-defiant expression upon his bold, well-marked ', {'entities': []}),
    ('white shirt, looped up upon one side, gave a glimpse of a huge knotty leg, scarred and torn ', {'entities': []}),
    ('with the scratches of brambles. With a bow to the Abbot, which had in it perhaps more ', {'entities': [(22, 30, 'ORG'), (50, 55, 'PERSON')]}),
    ('the private orisons of the Abbot's own household. His dark eyes glanced rapidly over the ', {'entities': [(27, 32, 'PERSON')]}),
    ('assembly, and finally settled with a grim and menacing twinkle upon the face of his accuser. ', {'entities': []})
]

exx3 =[
    ('Further on, at the edge of the woodland, he came upon a chapman and his wife, who sat upon ', {'entities': [(31, 39, 'ORG'), (56, 62, 'PERSON')]}),
    ('a fallen tree. He had put his pack down as a table, and the two of them were devouring a great ', {'entities': []}),
    ('pasty, and washing i down with some drink from a stone jar. The chapman broke a rough jest ', {'entities': [(65, 71, 'PERSON')]}),
    ('as he passed, and the woman called shrilly to Alleyne to come and join them, on which the ', {'entities': [(39, 41, 'PERSON'), (46, 52, 'PERSON')]}),
    ('man, turning suddenly from mirth to wrath, began to belabor her with his cudgel. Alleyne ', {'entities': [(81, 87, 'PERSON')]}),
    ('hastened on, lest he make more mischief, and his heart was heavy as lead within him. Look ', {'entities': []}),
    ('where he would, he seemed to see nothing but injustice and violence and the hardness of man to man. ', {'entities': []})
]

```

```

ex = []
ey=[]
ez=[]
exx = [
    ('At this order a lay-brother swung open the door, and two other lay-brothers entered leading ', {'entities': []}),
    ('headed, with a peculiar half-humorous, half-defiant expression upon his bold, well-marked ', {'entities': []}),
    ('white shirt, looped up upon one side, gave a glimpse of a huge knotty leg, scarred and torn ', {'entities': []}),
    ('with the scratches of brambles. With a bow to the Abbot, which had in it perhaps more ', {'entities': [(22, 30, 'ORG'), (50, 55, 'PERSON')]}),
    ('the private orisons of the Abbot's own household. His dark eyes glanced rapidly over the ', {'entities': [(27, 32, 'PERSON')]}),
    ('assembly, and finally settled with a grim and menacing twinkle upon the face of his accuser. ', {'entities': []})
]

exx3=[
    ('Further on, at the edge of the woodland, he came upon a chapman and his wife, who sat upon ', {'entities': [(31, 39, 'ORG'), (56, 62, 'PERSON')]}),
    ('a fallen tree. He had put his pack down as a table, and the two of them were devouring a Loading-- ', {'entities': []}),
    ('pasty, and washing i down with some drink from a stone jar. The chapman broke a rough jest ', {'entities': [(65, 71, 'PERSON')]}),
    ('as he passed, and the woman called shrilly to Alleyne to come and join them, on which the ', {'entities': [(35, 41, 'PERSON'), (46, 52, 'PERSON')]}),
    ('man, turning suddenly from mirth to wrath, began to labor her with his cudgel. Alleyne ', {'entities': [(81, 87, 'PERSON')]}),
    ('hastened on, lest he make more mischief, and his heart was heavy as lead within him. Look ', {'entities': []}),
    ('where he would, he seemed to see nothing but injustice and violence and the hardness of man to man. ', {'entities': []})
]

exx2 = [
    ('The night had already fallen, and the moon was shining between the rifts of ragged, drifting ', {'entities': []}),
    ('clouds, before Alleyne Edricson, footsore and weary from the unwonted exercise, found ', {'entities': [(15,30,'PERSON')]}),
    ('himself in front of the forest inn which stood upon the outskirts of Lyndhurst. The building ', {'entities': [(70,78,'LOC')]}),
    ('was long and low, standing back a little from the road, with two flambeaux blazing on either ', {'entities': []}),
    ('side of the door as a welcome to the traveller. From one window there thrust forth a long pole ', {'entities': []}),
    ('with a bunch of greenery tied to the end of it-a sign that liquor was to be sold within. As ', {'entities': []}),
    ('Alleyne walked up to it he perceived that it was rudely fashioned out of beams of wood, with ', {'entities': [(0,6,'PERSON')]}),
    ('twinkling lights all over where the glow from within shone through the chinks. The roof was ', {'entities': []}),
    ('poor and thatched; but in strange contrast to it there ran all along under the eaves a line of ', {'entities': []}),
    ('wooden shields, most gorgeously painted with chevron, bend, and saltire, and every heraldic ', {'entities': [(45,52,'ORG')]}),
    ('device. By the door a horse stood tethered, the ruddy glow beating strongly upon his brown ', {'entities': []}),
    ('head and patient eyes, while his body stood back in the shadow.' ,{'entities': []})
]

fun(t2,ex,exx)
fun(t4,e2,exx2)
fun(t3,ey,exx3)

```

```

[['At this order a lay-brother swung open the door, and two other lay-brothers entered leading ', {'entities': [(53, 56, 'CARDINAL')]}]
Precision 0.1429      Recall 0.3333      F-score 0.2000
[['The night had already fallen, and the moon was shining between the rifts of ragged, drifting ', {'entities': []}], ('clouds, before
Precision 0.2500      Recall 1.0000      F-score 0.4000
[['Further on, at the edge of the woodland, he came upon a chapman and his wife, who sat upon ', {'entities': []}], ('a fallen tree. H
Precision 0.3333      Recall 0.6667      F-score 0.4444

```


To evaluate our model we First Manully labelled 3 Texts from the book taken by us and then we created a function name fun and evaluate to then label our data with NER Tags using spacy and and then evaluate the Precision Recall and F value using it.

**The Second Part of this project aimed at Generating TF-IDF Vector for each Chapter and then Calculating which two chapter are most similar.**



## Libraries Used:

```
import re
from collections import defaultdict
```



```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

We generated TF-IDF vectors and Then created a cosine similarity matrix of chapters and then Traversed the whole matrix to find the two most similar Chapters

The code uses scikit-learn's 'TfidfVectorizer' to convert the text data ('chapters') into a TF-IDF matrix ('tfidf\_matrix'). This matrix represents the chapters as vectors in a high-dimensional space, emphasizing unique terms' importance across chapters.

Using 'cosine\_similarity' from scikit-learn, the code computes the pairwise cosine similarity between all chapters, generating a similarity matrix ('sim\_mat') where each element represents the similarity between two chapters.

This loop iterates through the similarity matrix ('sim\_mat') to find the indices of the two most similar chapters by comparing pairwise similarities. It updates 'max\_sim' and 'index' when it finds a higher similarity score.

**Visualization** is done using Seaborn and Matplotlib we create a heatmap visualization of the cosine similarity matrix for better visualization of chapter similarities.

```

with open('book1.txt', 'r') as file:
    data = file.read()

# Assuming chapters are separated by 'Chapter {number}'
chapters= re.split(r'Chapter', data)[39:]
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(chapters)
sim_mat = cosine_similarity(tfidf_matrix)

max_sim = 0
index = (0, 0)

# Compare all pairs of documents to find the most similar
for i in range(len(chapters)):
    for j in range(i + 1, len(chapters)):
        sim = sim_mat[i, j]
        if sim > max_sim:
            max_sim = sim
            index = (i, j)

# Print the similarity matrix
print("Cosine Similarity Matrix:")
print(sim_mat)

# Print the indices of the two most similar documents
print("\nIndices of the Two Most Similar Documents:", index)

sns.heatmap(sim_mat,cmap='YlGnBu')
plt.title('Cosine Similarity Matrix')
plt.show()

```

The Output of the following Code can be seen here:

Cosine Similarity Matrix:

```
[[1.          0.87995037 0.8152593 ... 0.87519008 0.8564495 0.84754407]
 [0.87995037 1.          0.82121287 ... 0.8712782 0.84379295 0.8468052 ]
 [0.8152593  0.82121287 1.          ... 0.82736092 0.8244328 0.83527369]
 ...
 [0.87519008 0.8712782 0.82736092 ... 1.          0.91434189 0.89284886]
 [0.8564495  0.84379295 0.8244328 ... 0.91434189 1.          0.87930651]
 [0.84754407 0.8468052 0.83527369 ... 0.89284886 0.87930651 1.          ]]
```

Indices of the Two Most Similar Documents: (34, 35)

The gradient Table looks like this

Indices of the Two Most Similar Documents: (34, 35)

