Natural Language Processing

CSE-3201

Project Report

Team: Text Techies

Submitted by:
Piyush Sharma(21ucs150)
Shreshtha Gupta (21ucs196)
Abhay Gupta (21UCS002)

Book: "Inferno" By Dan Brown

GitHub Link:
https://github.com/Piyush9843/NLP_Project

Round 1

Objective

The goal of this text analysis report is to thoroughly examine a provided text document in.txt format ('Inferno' by Dan Brown).

Text preprocessing is used to clean and prepare the text, tokenization is used to split it down into individual words, and stop words are removed to focus on significant material. To discover common words, a frequency distribution analysis is performed, and a word cloud is generated for visualization. The distribution of tags is studied after doing part-of-speech tagging using a Penn Treebank tag set.

In addition, a bi-gram probability table with stop words is constructed for the book's major chapter. The "Shannon game" is played in a separate chapter by filling in blanks with bi-gram probabilities and accuracy.

Data Description

The book used for this project is '**Inferno**' by Dan Brown (in .txt Format).

Title: **Inferno**

Author: **Dan Brown**

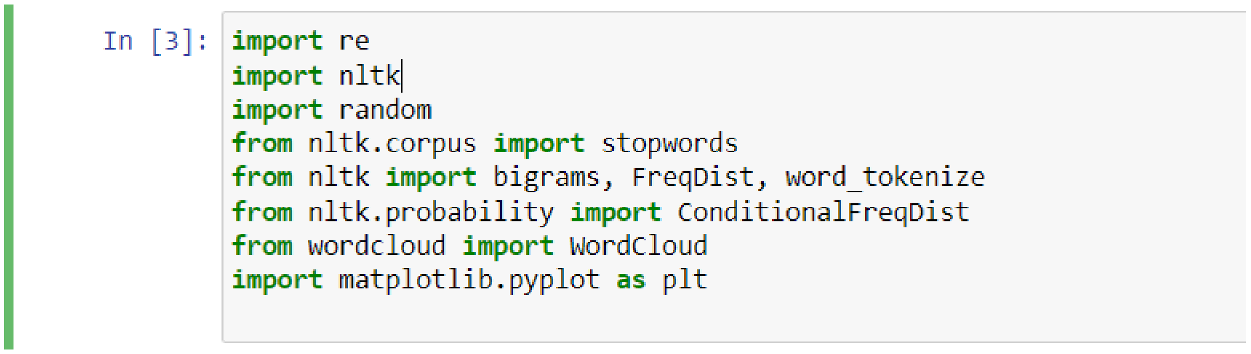
Number of words: **183317**

Importing Libraries

To fulfill our goals, we must import necessary libraries into the code and perform essential operations on our textbook.

The required Libraries are:

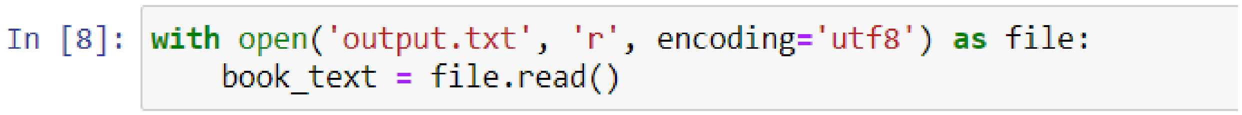
re, nltk, random, wordcloud, matplotlib, stopwords, bigrams, FreqDist, word_tokenize, ConditionalFreqDist



```
In [3]: import re
        import nltk
        import random
        from nltk.corpus import stopwords
        from nltk import bigrams, FreqDist, word_tokenize
        from nltk.probability import ConditionalFreqDist
        from wordcloud import WordCloud
        import matplotlib.pyplot as plt
```

Importing Book

We import our book which is in .txt format.



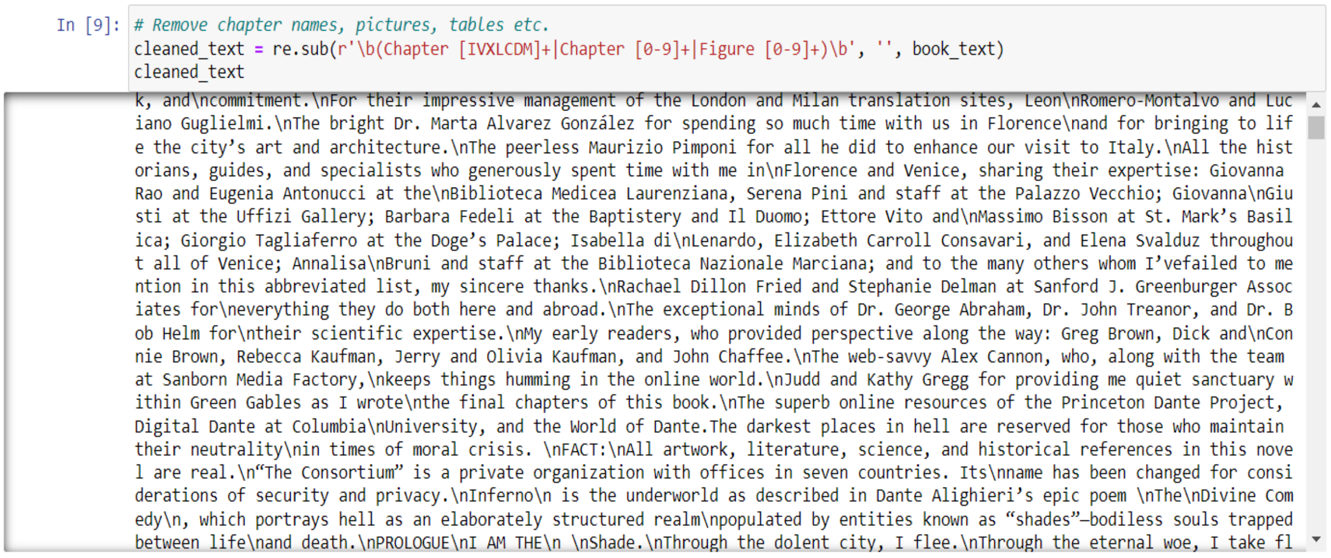
```
In [8]: with open('output.txt', 'r', encoding='utf8') as file:
            book_text = file.read()
```

Text Pre-Processing

Text pre-processing is an important stage in the data preparation pipeline, as it improves the quality and usability of textual data. We developed many essential pre-processing stages in the framework of our project to ensure the text is clean and ready for analysis.

One of the most important steps entailed removing running section or chapter labels, which frequently crowded the material and could potentially contribute to noise in our study. To accomplish this, we used regular expressions to discover and delete recurrent patterns. We also deleted unnecessary components like photos and tables because they didn't contribute to the textual analysis we were doing.

We aimed to build a refined text corpus that is easier to deal with through these pre-processing techniques.

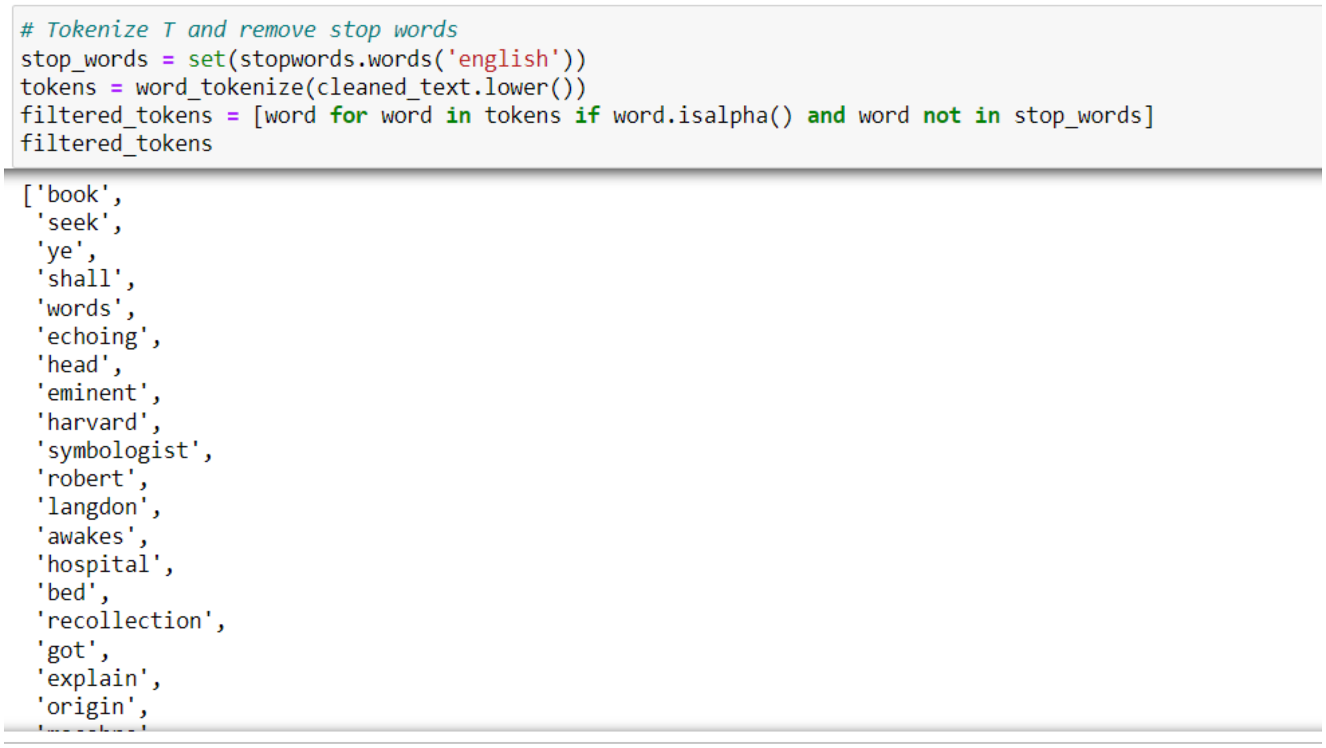


```
In [9]: # Remove chapter names, pictures, tables etc.
        cleaned_text = re.sub(r'\b(Chapter [IVXLCDM]+|Chapter [0-9]+|Figure [0-9]+)\b', '', book_text)
        cleaned_text
```

```
k, and\ncommitment.\nFor their impressive management of the London and Milan translation sites, Leon\nRomero-Montalvo and Luc
iano Guglielmi.\nThe bright Dr. Marta Alvarez González for spending so much time with us in Florence\nand for bringing to lif
e the city's art and architecture.\nThe peerless Maurizio Pimponi for all he did to enhance our visit to Italy.\nAll the hist
orians, guides, and specialists who generously spent time with me in\nFlorence and Venice, sharing their expertise: Giovanna
Rao and Eugenia Antonucci at the\nBiblioteca Medicea Laurenziana, Serena Pini and staff at the Palazzo Vecchio; Giovanna\nGiu
sti at the Uffizi Gallery; Barbara Fedeli at the Baptistery and Il Duomo; Ettore Vito and\nMassimo Bisson at St. Mark's Basil
ica; Giorgio Tagliaferro at the Doge's Palace; Isabella di\nLenardo, Elizabeth Carroll Consavari, and Elena Svalduz throughou
t all of Venice; Annalisa\nBruni and staff at the Biblioteca Nazionale Marciana; and to the many others whom I'vefailed to me
ntion in this abbreviated list, my sincere thanks.\nRachael Dillon Fried and Stephanie Delman at Sanford J. Greenburger Assoc
iates for\neverything they do both here and abroad.\nThe exceptional minds of Dr. George Abraham, Dr. John Treanor, and Dr. B
ob Helm for\ntheir scientific expertise.\nMy early readers, who provided perspective along the way: Greg Brown, Dick and\nCon
nie Brown, Rebecca Kaufman, Jerry and Olivia Kaufman, and John Chaffee.\nThe web-savvy Alex Cannon, who, along with the team
at Sanborn Media Factory,\nkeeps things humming in the online world.\nJudd and Kathy Gregg for providing me quiet sanctuary w
ithin Green Gables as I wrote\nthe final chapters of this book.\nThe superb online resources of the Princeton Dante Project,
Digital Dante at Columbia\nUniversity, and the World of Dante.The darkest places in hell are reserved for those who maintain
their neutrality\nin times of moral crisis. \nFACT:\nAll artwork, literature, science, and historical references in this nove
l are real.\n"The Consortium" is a private organization with offices in seven countries. Its\nname has been changed for consi
derations of security and privacy.\nInferno\n is the underworld as described in Dante Alighieri's epic poem \nThe\nDivine Com
edy\n, which portrays hell as an elaborately structured realm\npopulated by entities known as "shades"—bodiless souls trapped
between life\nand death.\nPROLOGUE\nI AM THE\n \nShade.\nThrough the dolent city, I flee.\nThrough the eternal woe, I take fl
```

Tokenizing Our Dataset

- **Stopword Removal**: We first created a set of English stopwords using *stopwords.word()* function of the NLTK library.
- **Text Tokenization**: We tokenized the cleaned text, breaking it into individual words using *word_tokenize()* function.
- **Lowercasing**: To ensure consistency, we converted all tokens to lowercase using *cleaned_text.lower()* function.
- **Filtering Non-Alphabetic Words**: We removed tokens that contained non-alphabetic characters, retaining only words.
- **Eliminating Stopwords**: We filtered out stopwords, ensuring that the resulting 'filtered_tokens' list contained only meaningful and relevant words for subsequent text analysis.

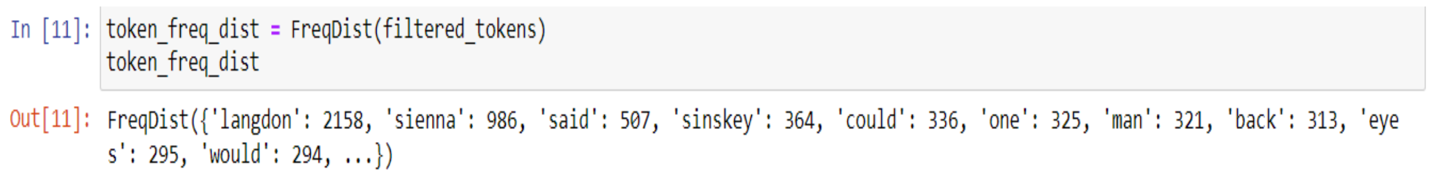


```python
# Tokenize T and remove stop words
stop_words = set(stopwords.words('english'))
tokens = word_tokenize(cleaned_text.lower())
filtered_tokens = [word for word in tokens if word.isalpha() and word not in stop_words]
filtered_tokens
```

```
['book',
 'seek',
 'ye',
 'shall',
 'words',
 'echoing',
 'head',
 'eminent',
 'harvard',
 'symbologist',
 'robert',
 'langdon',
 'awakes',
 'hospital',
 'bed',
 'recollection',
 'got',
 'explain',
 'origin',
```

Calculating The Frequency Of Tokens

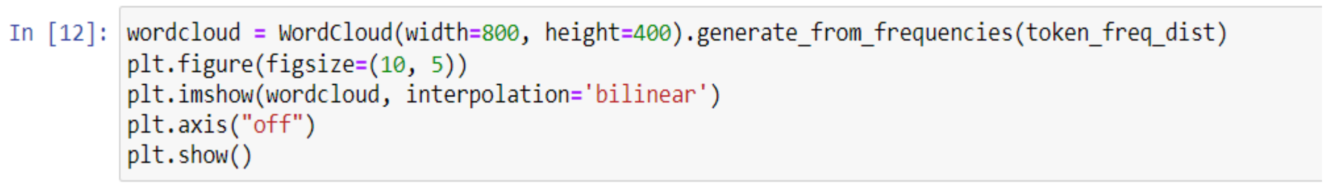
- **Frequency Distribution:** We used the NLTK library to create a frequency distribution of the 'filtered_tokens.' A frequency distribution counts the occurrences of each unique word in the text.
- **Insights into Word Occurrence:** By generating this 'token_freq_dist,' we gained valuable insights into how often different words appear in our pre-processed text data.
- **Useful for Analysis:** This frequency distribution is a foundational step for various text analysis tasks, such as identifying the most common words, understanding the vocabulary, or even spotting trends in the text corpus.



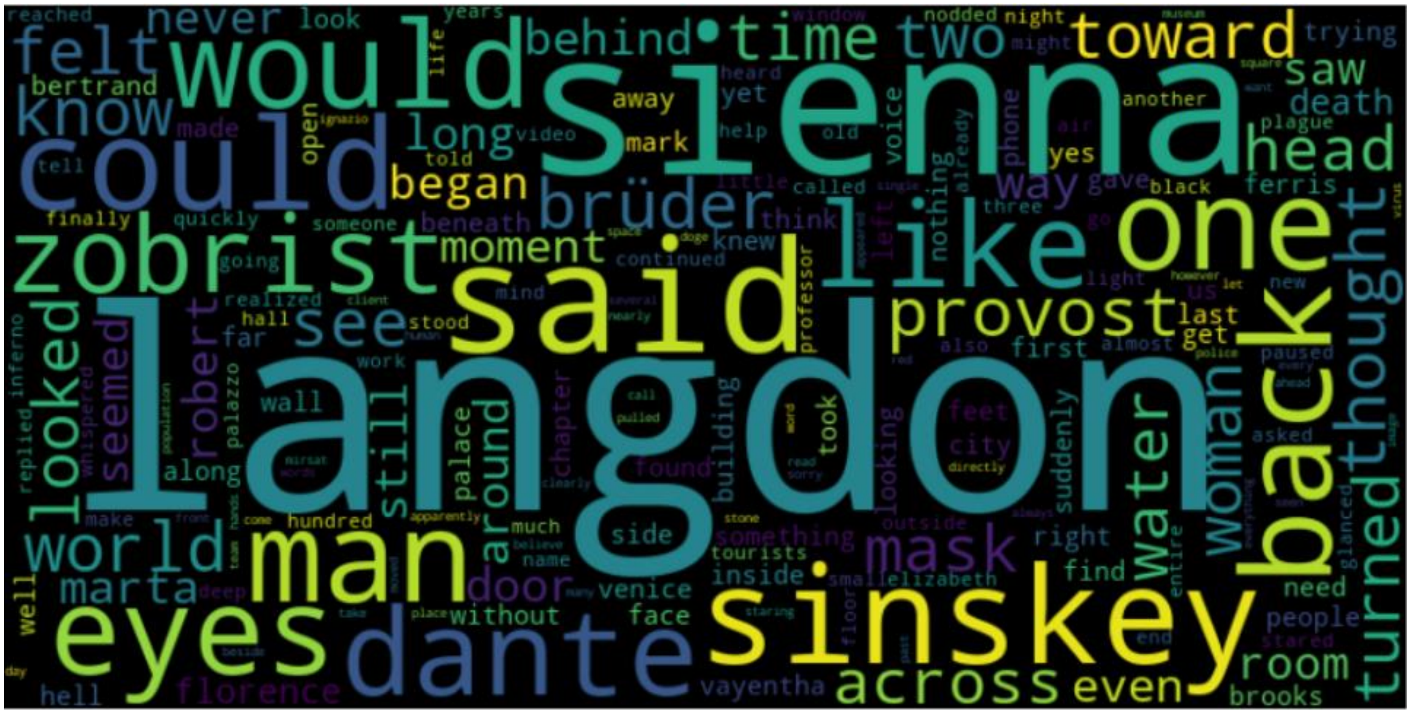
```
In [11]: token_freq_dist = FreqDist(filtered_tokens)
         token_freq_dist

Out[11]: FreqDist({'langdon': 2158, 'sienna': 986, 'said': 507, 'sinskey': 364, 'could': 336, 'one': 325, 'man': 321, 'back': 313, 'eye
         s': 295, 'would': 294, ...})
```

Creating Word Cloud

- **Word Cloud Creation:** We utilized the WordCloud library to create a visual representation of the most frequently occurring words in our text data. The 'WordCloud' object was instantiated with specific width and height parameters, defining the visual dimensions of the word cloud.
- **Frequency-Based Generation:** The word cloud was generated from the frequency distribution we previously created ('token_freq_dist'). This means that words that occurred more frequently in the text would be displayed more prominently in the word cloud.
- **Visualization Setup:** We configured the visualization by setting up a plot with specific dimensions (10x5 inches) to ensure clarity and readability.
- **Word Cloud Rendering:** The 'wordcloud' was then rendered within the plot using '*imshow*,' and 'interpolation' was set to 'bilinear' for smoother visualization.
- **Aesthetic Presentation:** We removed axis labels ('*plt.axis("off")*') to create a clean and aesthetically pleasing word cloud image.
- **Display:** Finally, the word cloud was displayed for visual exploration and analysis.

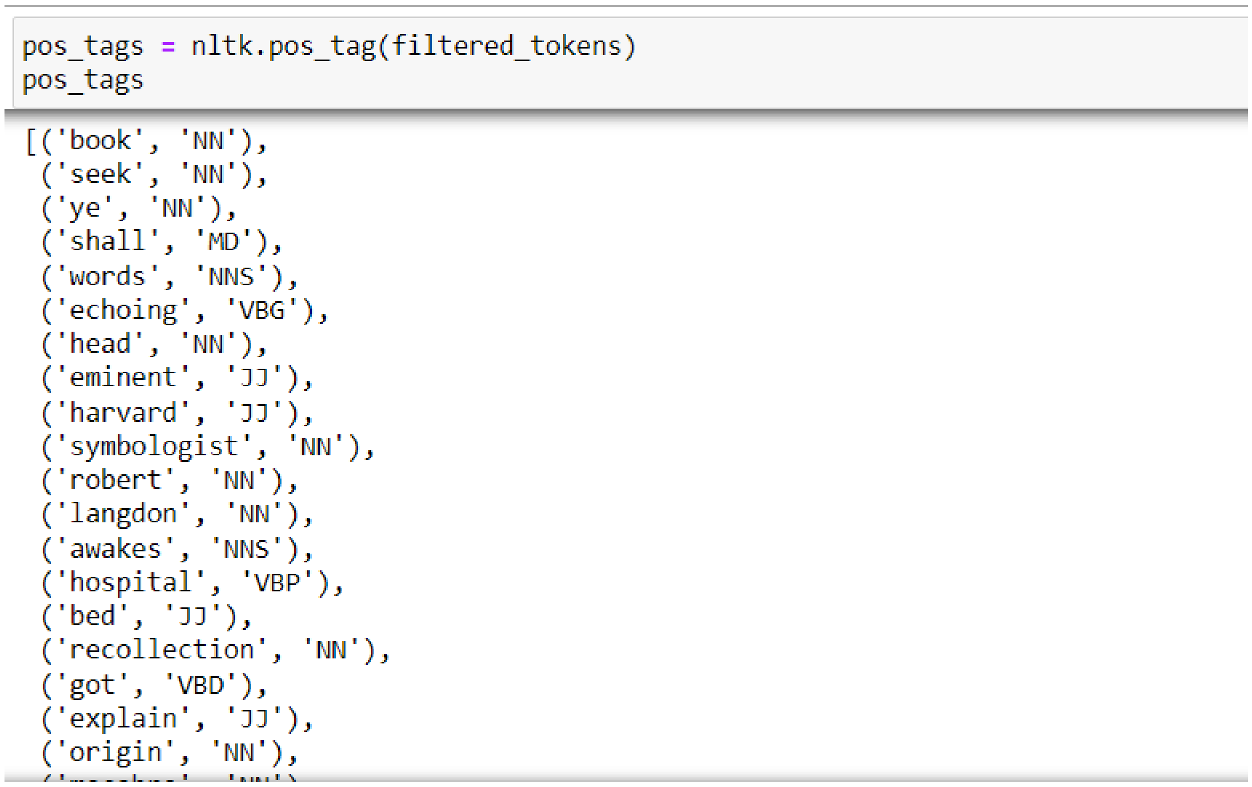


```
In [12]: wordcloud = WordCloud(width=800, height=400).generate_from_frequencies(token_freq_dist)
         plt.figure(figsize=(10, 5))
         plt.imshow(wordcloud, interpolation='bilinear')
         plt.axis("off")
         plt.show()
```



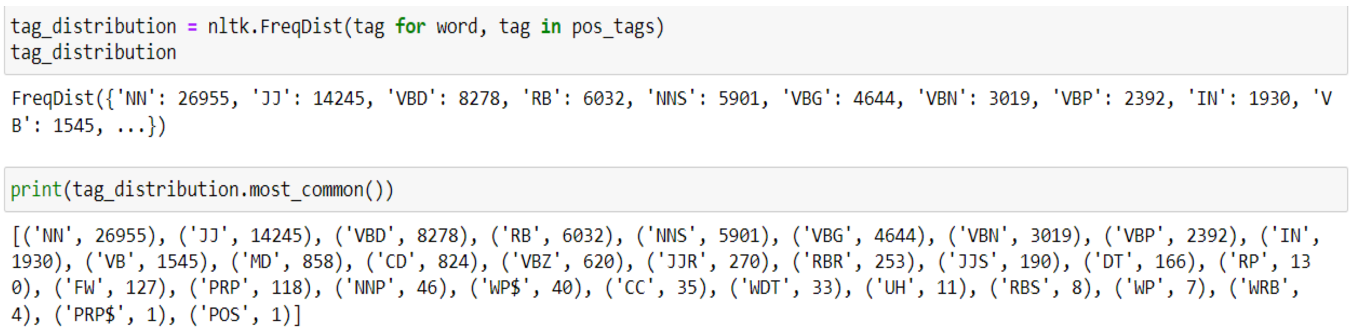
POS Tagging

- **POS Tagging:** We applied POS tagging to our 'filtered_tokens' using NLTK (*pos_tag*) library, which assigned grammatical categories (e.g., noun, verb) to each word. Pos_tag Function of nltk library uses **Penn Treebank tagset** .
- **Tag Frequency Distribution:** The 'tag_distribution' was created to count the occurrences of different grammatical categories in the text.
- **Insights:** This analysis gives insights into the text's grammatical structure, which is valuable for understanding language usage and aids in various text analysis tasks.
- **Result Display:** We printed the most common POS tags to provide a quick overview of the distribution of grammatical categories in the text.



```
pos_tags = nltk.pos_tag(filtered_tokens)
pos_tags
```

```
[('book', 'NN'),
 ('seek', 'NN'),
 ('ye', 'NN'),
 ('shall', 'MD'),
 ('words', 'NNS'),
 ('echoing', 'VBG'),
 ('head', 'NN'),
 ('eminent', 'JJ'),
 ('harvard', 'JJ'),
 ('symbologist', 'NN'),
 ('robert', 'NN'),
 ('langdon', 'NN'),
 ('awakes', 'NNS'),
 ('hospital', 'VBP'),
 ('bed', 'JJ'),
 ('recollection', 'NN'),
 ('got', 'VBD'),
 ('explain', 'JJ'),
 ('origin', 'NN'),
```



```
tag_distribution = nltk.FreqDist(tag for word, tag in pos_tags)
tag_distribution
```

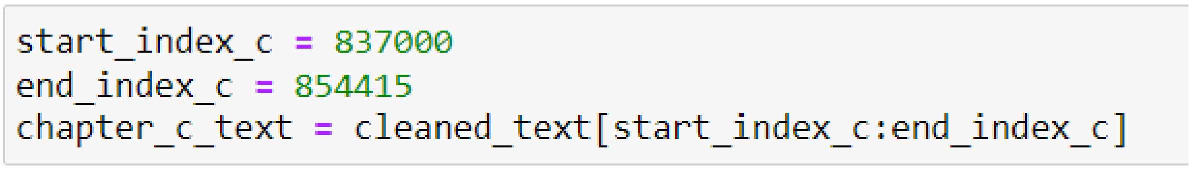
```
FreqDist({'NN': 26955, 'JJ': 14245, 'VBD': 8278, 'RB': 6032, 'NNS': 5901, 'VBG': 4644, 'VBN': 3019, 'VBP': 2392, 'IN': 1930, 'V
B': 1545, ...})
```

```
print(tag_distribution.most_common())
```

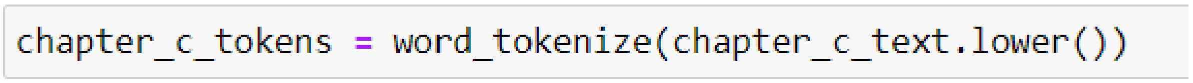
```
[('NN', 26955), ('JJ', 14245), ('VBD', 8278), ('RB', 6032), ('NNS', 5901), ('VBG', 4644), ('VBN', 3019), ('VBP', 2392), ('IN',
1930), ('VB', 1545), ('MD', 858), ('CD', 824), ('VBZ', 620), ('JJR', 270), ('RBR', 253), ('JJS', 190), ('DT', 166), ('RP', 13
0), ('FW', 127), ('PRP', 118), ('NNP', 46), ('WP$', 40), ('CC', 35), ('WDT', 33), ('UH', 11), ('RBS', 8), ('WP', 7), ('WRB',
4), ('PRP$', 1), ('POS', 1)]
```

Extracting the largest chapter and Tokenization

- **Chapter Extraction:** We extracted the largest chapter, namely 'Chapter C,' from our pre-processed text data. We defined the start and end indices for this chapter within the 'cleaned_text.'
- In our case, The largest chapter is Chapter 100, and Starting index = 837000, and End Index = 854415
- **Text Tokenization:** After extracting 'Chapter C,' we tokenized the text within this specific chapter, converting it to lowercase. This process prepares the text for further analysis within this particular section.
- This action isolates 'Chapter C' for in-depth analysis or focused investigations, as needed for our project.



```python
start_index_c = 837000
end_index_c = 854415
chapter_c_text = cleaned_text[start_index_c:end_index_c]
```



```python
chapter_c_tokens = word_tokenize(chapter_c_text.lower())
```

Bigram for the largest chapter

- **Bigram Extraction:** We created bigrams, which are pairs of consecutive words, from the tokenized text within 'Chapter C’. This step allows us to analyze word relationships and co-occurrences in this specific section of the text.
- **Bigram Frequency Distribution:** We generated a frequency distribution ('bi_gram_freq_c') to count the occurrences of each unique bigram within 'Chapter C.' This distribution provides insights into the most common word pairs in this chapter.
- **Conditional Frequency Distribution:** Additionally, we calculated conditional frequencies ('bi_gram_probabilities_c') of bigrams. This

means we are not only tracking how often a bigram occurs but also how often the second word follows the first word. It can offer valuable insights into word associations and contextual relationships within 'Chapter C.'

```
bi_grams_c = list(bigrams(chapter_c_tokens))
bi_gram_freq_c = FreqDist(bi_grams_c)
bi_gram_probabilities_c = ConditionalFreqDist(bi_grams_c)
```

```
bi_grams_c
```

```
[('transpiring', 'at'),
 ('at', 'the'),
 ('the', 'cistern'),
 ('cistern', '.'),
 ('.', 'by'),
 ('by', 'now'),
 ('now', ','),
 (',', 'he'),
 ('he', 'suspected'),
 ('suspected', ','),
 (',', 'sinskey'),
 ('sinskey', 'and'),
 ('and', 'the'),
 ('the', 'srs'),
 ('srs', 'team'),
 ('team', 'had'),
 ('had', 'realized'),
 ('realized', 'that'),
 ('that', 'they'),
```

```
bi_gram_freq_c
```

```
FreqDist({('.', '“'): 59, ('”', '“'): 24, (',', '”'): 23, ('’', 's'): 23, ('”', 'langdon'): 22, ('’', 't'): 20, ('”', 'she'): 1
8, ('“', 'i'): 15, ('.', 'he'): 14, ('of', 'the'): 14, ...})
```

➢ We will extract any other chapter (in this case consider Ch 50) and tokenize it as it is required for playing Shannon Game.

```
#Extracting any randomm chapter
start_index_other = 391725
end_index_other = 396594

chapter_other_text = cleaned_text[start_index_other:end_index_other]
```

```
chapter_other_tokens = word_tokenize(chapter_other_text.lower())
```

Shannon Game

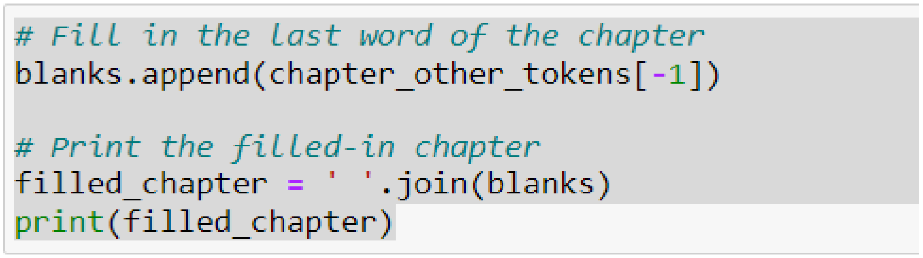
- **Shannon Game:** We used the "Shannon Game" text modification technique within the 'chapter_other' content. The Shannon Game includes blanking out specific words in a text to see how they affect readability and comprehension. This can be a useful tool for investigating how changes in language affect comprehension.
- **Initialization:** We created an empty list called 'blanks' to hold the updated content.
- **Iteration:** We went through the 'chapter_other_tokens' to look at each word and how it related to the next word in the text.
- **Stopword Handling:** If the current word is a stopword, it is simply added to the 'blanks' list. This prevents stopwords from being blanked out, which are often less informative words like "the" and "and."
- **Bigram Probability Check:** In 'bi_gram_probabilities_c,' we calculated a random probability if the current word and the following word form a known bigram. If the generated random number is smaller than the likelihood of the next word in the bigram following the current word, we added the current word to the 'blanks' list. Otherwise, we substituted "___" for a blank to simulate a word omission.
- **Else Clause:** If the current word and the following word do not create a recognized bigram, we simply add the current word to the 'blanks' list.

```
blanks = []
for i in range(len(chapter_other_tokens) - 1):
    current_word = chapter_other_tokens[i]
    next_word = chapter_other_tokens[i + 1]
    if current_word in stop_words:
        blanks.append(current_word)
    elif (current_word, next_word) in bi_gram_probabilities_c:
        if random.random() < bi_gram_probabilities_c[current_word].freq(next_word):
            blanks.append(current_word)
        else:
            blanks.append("___")
    else:
        blanks.append(current_word)

blanks
```

```
['et',
 'passages',
 'tour',
 '.',
 'the',
 'duke',
 'of',
 'athens',
 'stairway',
 '.',
 'the',
 'sound',
 'of',
 'running',
```

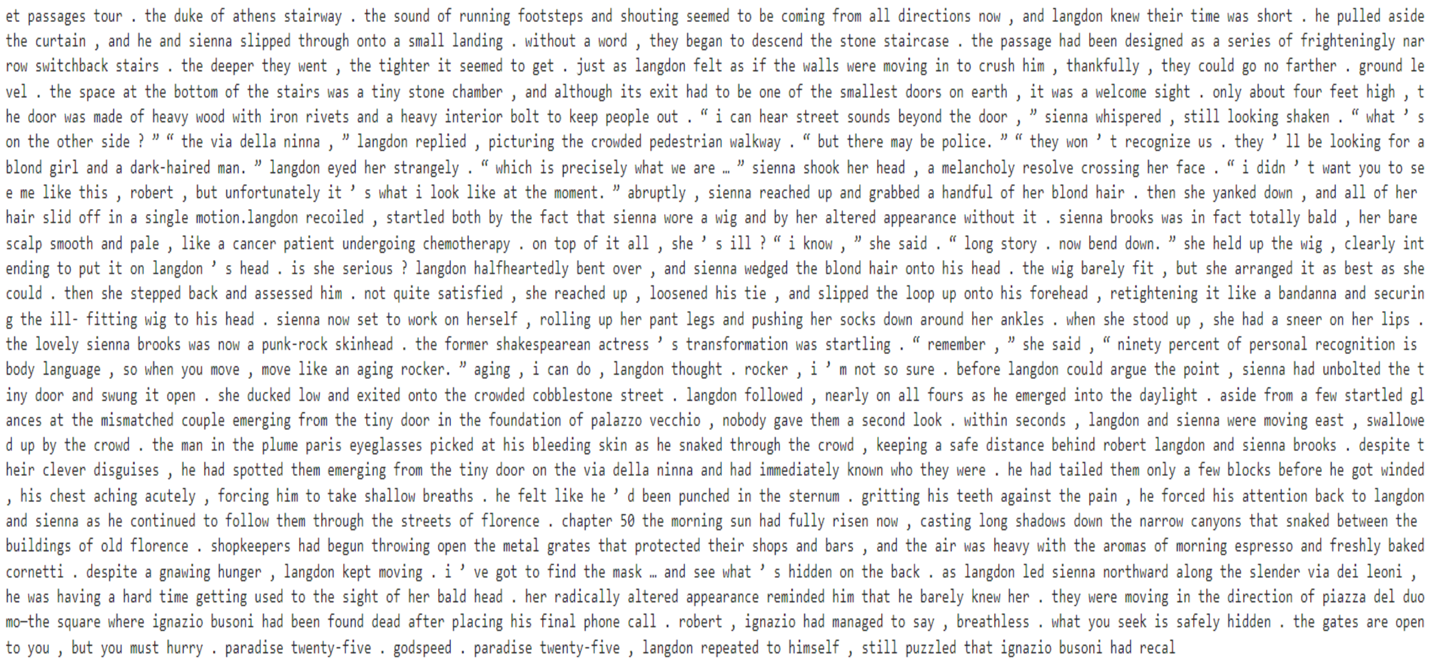
Fill in the blanks game with bi-gram probabilities

- After playing the Shannon Game and filling in blanks across 'chapter_other,' we finished the chapter by adding the final word. This stage guarantees that the chapter is coherent and ends on time.
- We rebuilt the amended 'chapter_other' content by combining the components in the 'blanks' list with a space as a separator.
- Finally, we printed the filled-in chapter so that we could examine the text after using the Shannon Game. This stage allows us to examine the influence of the word changes on the readability and overall narrative flow of the text.



```
# Fill in the last word of the chapter
blanks.append(chapter_other_tokens[-1])

# Print the filled-in chapter
filled_chapter = ' '.join(blanks)
print(filled_chapter)
```

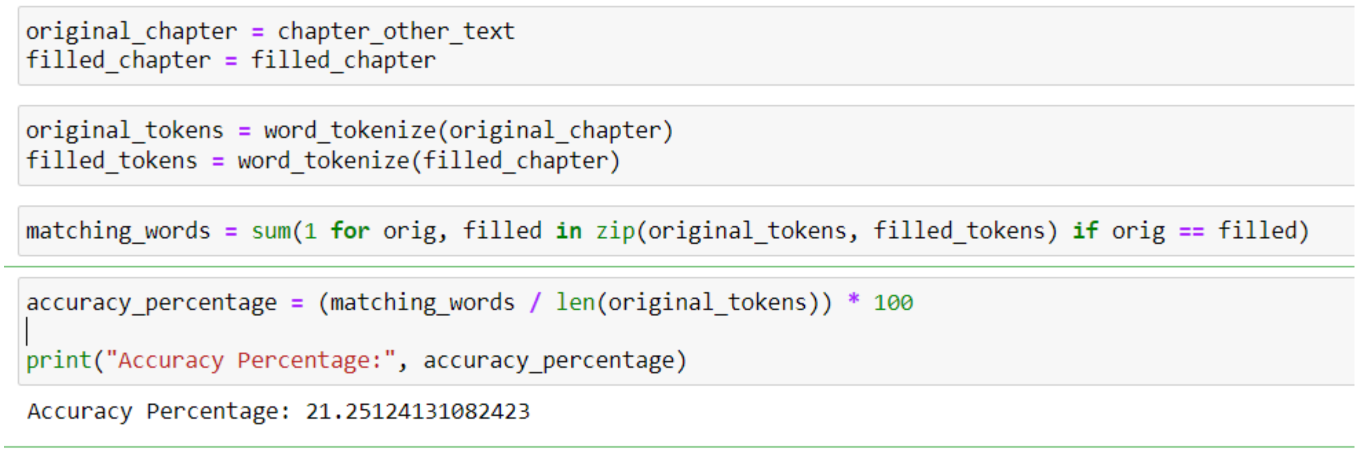
Output



```
et passages tour . the duke of athens stairway . the sound of running footsteps and shouting seemed to be coming from all directions now , and langdon knew their time was short . he pulled aside
the curtain , and he and sienna slipped through onto a small landing . without a word , they began to descend the stone staircase . the passage had been designed as a series of frighteningly nar
row switchback stairs . the deeper they went , the tighter it seemed to get . just as langdon felt as if the walls were moving in to crush him , thankfully , they could go no farther . ground le
vel . the space at the bottom of the stairs was a tiny stone chamber , and although its exit had to be one of the smallest doors on earth , it was a welcome sight . only about four feet high , t
he door was made of heavy wood with iron rivets and a heavy interior bolt to keep people out . “ i can hear street sounds beyond the door , ” sienna whispered , still looking shaken . “ what ’ s
on the other side ? ” “ the via della ninna , ” langdon replied , picturing the crowded pedestrian walkway . “ but there may be police. ” “ they won ’ t recognize us . they ’ ll be looking for a
blond girl and a dark-haired man. ” langdon eyed her strangely . “ which is precisely what we are … ” sienna shook her head , a melancholy resolve crossing her face . “ i didn ’ t want you to se
e me like this , robert , but unfortunately it ’ s what i look like at the moment. ” abruptly , sienna reached up and grabbed a handful of her blond hair . then she yanked down , and all of her
hair slid off in a single motion.langdon recoiled , startled both by the fact that sienna wore a wig and by her altered appearance without it . sienna brooks was in fact totally bald , her bare
scalp smooth and pale , like a cancer patient undergoing chemotherapy . on top of it all , she ’ s ill ? “ i know , ” she said . “ long story . now bend down. ” she held up the wig , clearly int
ending to put it on langdon ’ s head . is she serious ? langdon halfheartedly bent over , and sienna wedged the blond hair onto his head . the wig barely fit , but she arranged it as best as she
could . then she stepped back and assessed him . not quite satisfied , she reached up , loosened his tie , and slipped the loop up onto his forehead , retightening it like a bandanna and securin
g the ill- fitting wig to his head . sienna now set to work on herself , rolling up her pant legs and pushing her socks down around her ankles . when she stood up , she had a sneer on her lips .
the lovely sienna brooks was now a punk-rock skinhead . the former shakespearean actress ’ s transformation was startling . “ remember , ” she said , “ ninety percent of personal recognition is
body language , so when you move , move like an aging rocker. ” aging , i can do , langdon thought . rocker , i ’ m not so sure . before langdon could argue the point , sienna had unbolted the t
iny door and swung it open . she ducked low and exited onto the crowded cobblestone street . langdon followed , nearly on all fours as he emerged into the daylight . aside from a few startled gl
ances at the mismatched couple emerging from the tiny door in the foundation of palazzo vecchio , nobody gave them a second look . within seconds , langdon and sienna were moving east , swallowe
d up by the crowd . the man in the plume paris eyeglasses picked at his bleeding skin as he snaked through the crowd , keeping a safe distance behind robert langdon and sienna brooks . despite t
heir clever disguises , he had spotted them emerging from the tiny door on the via della ninna and had immediately known who they were . he had tailed them only a few blocks before he got winded
, his chest aching acutely , forcing him to take shallow breaths . he felt like he ’ d been punched in the sternum . gritting his teeth against the pain , he forced his attention back to langdon
and sienna as he continued to follow them through the streets of florence . chapter 50 the morning sun had fully risen now , casting long shadows down the narrow canyons that snaked between the
buildings of old florence . shopkeepers had begun throwing open the metal grates that protected their shops and bars , and the air was heavy with the aromas of morning espresso and freshly baked
cornetti . despite a gnawing hunger , langdon kept moving . i ’ ve got to find the mask … and see what ’ s hidden on the back . as langdon led sienna northward along the slender via dei leoni ,
he was having a hard time getting used to the sight of her bald head . her radically altered appearance reminded him that he barely knew her . they were moving in the direction of piazza del duo
mo—the square where ignazio busoni had been found dead after placing his final phone call . robert , ignazio had managed to say , breathless . what you seek is safely hidden . the gates are open
to you , but you must hurry . paradise twenty-five . godspeed . paradise twenty-five , langdon repeated to himself , still puzzled that ignazio busoni had recal
```

Accuracy Analysis

The accuracy percentage is *21.25124131082423%.* This percentage depicts the Shannon game's level of accuracy, which measures how closely the generated text matches the original text**.**



```
original_chapter = chapter_other_text
filled_chapter = filled_chapter
```

```
original_tokens = word_tokenize(original_chapter)
filled_tokens = word_tokenize(filled_chapter)
```

```
matching_words = sum(1 for orig, filled in zip(original_tokens, filled_tokens) if orig == filled)
```

```
accuracy_percentage = (matching_words / len(original_tokens)) * 100

print("Accuracy Percentage:", accuracy_percentage)
```

```
Accuracy Percentage: 21.25124131082423
```

Round 2

Part 1

Objective

This task is to create and assess an entity recognition system based on the entity types defined in Fig. 22.1. The process consists of two main steps: first, identifying all entities within the passages that are given, and second, classifying these identified entities into the appropriate types as defined in Fig. 22.1. Several evaluations will be carried out to gauge how well this method performs. First, random passages from the source material will be manually labeled to identify entities and their types, providing a ground truth dataset for comparison. This manual labeling process will be done three times to guarantee a robust evaluation. Finally, the performance will be measured using F1 scores, a metric that takes into account both precision and recall of the identified entities and

Importing Libraries

To fulfill our goals, we must import necessary libraries into the code and perform essential operations on our textbook.

The required Libraries are:

Re, nltk, matplotlib, sklearn, numpy

```
import re
import nltk
import matplotlib.pyplot as plt
from nltk.tokenize import word_tokenize
from nltk.tag import pos_tag
from nltk.chunk import ne_chunk
from sklearn.metrics import f1_score
```

Performing NER

```python
bookNER = set()
for sent in nltk.sent_tokenize(cleaned_text):
    for chunk in nltk.ne_chunk(nltk.pos_tag(nltk.word_tokenize(sent))):
        if hasattr(chunk, 'label'):
            bookNER.add((' '.join(c[0] for c in chunk), chunk.label()))

bookNER
```

This code is a Python snippet that uses the Natural Language Toolkit (NLTK) package to perform Named Entity Recognition (NER). Its goal is to find named entities in a text and classify them according to labels that have been established.

Output:

```
{('VIA', 'ORGANIZATION'),
 ('Firenze', 'PERSON'),
 ('Field', 'GPE'),
 ('Carnevale', 'ORGANIZATION'),
 ('Solublon', 'PERSON'),
 ('Borges', 'GPE'),
 ('Tonight', 'GPE'),
 ('See', 'PERSON'),
 ('Philippines', 'ORGANIZATION'),
 ('RNA', 'ORGANIZATION'),
 ('Sorry', 'PERSON'),
 ('U.S.', 'GPE'),
 ('NO', 'ORGANIZATION'),
 ('Magic', 'GPE'),
 ('Amitriptyline', 'PERSON'),
 ('Brooks', 'GPE'),
 ('Pitti', 'ORGANIZATION'),
 ('Gypsy', 'PERSON'),
 ('Part', 'ORGANIZATION'),
 ('Hecate', 'GPE'),
 ('Keep Grandma', 'PERSON'),
 ('Don', 'PERSON'),
 ('Seven Towers', 'ORGANIZATION'),
 ('La', 'PERSON'),
 ('Neither', 'PERSON'),
 ('Palazzo Vecchio', 'GPE'),
 ('Death', 'PERSON'),
 ('New Mexico', 'GPE'),
 ('Signorina', 'PERSON'),
 ('FS', 'ORGANIZATION'),
```

We have performed the NER for full book.

Let's break down the code's functionality step by step:

- **Text Processing:**

  The code starts by processing a text document, presumably after some cleaning, into sentences using NLTK's *sent_tokenize* function. This step aims to break the text into individual sentences for further analysis.

- **Tokenization and Part-of-Speech Tagging:**

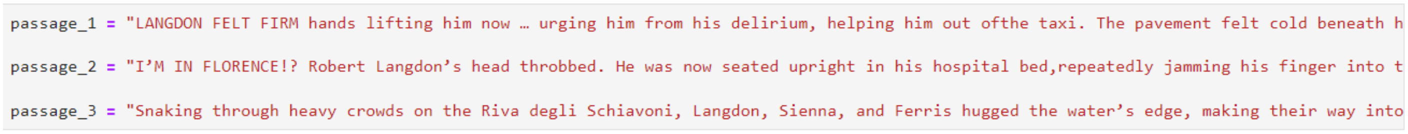
  Within each sentence, the code tokenizes the text into words using *word_tokenize* and assigns part-of-speech (POS) tags to each word using *pos_tag*. This step is crucial in understanding the grammatical structure of the text.

- **Named Entity Recognition (NER):**
  The code employs NLTK's *ne_chunk* function, which utilizes a pre-trained named entity chunker, to identify and extract chunks that represent named entities from the POS-tagged text. It detects phrases that correspond to entities such as persons, organizations, locations, etc.

- **Entity Labeling and Storage:**
  As the code identifies chunks with labels indicative of named entities, it stores this information in a set called *bookNER*. This set is composed of tuples, each containing the identified entity text and its corresponding entity label.

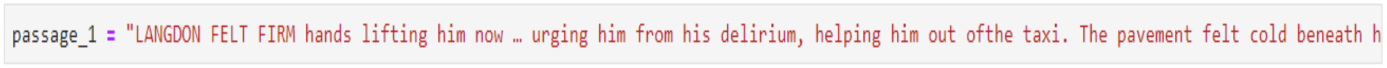
Evaluation

In order to assess the efficacy of a Named Entity Recognition (NER) system, we will be using particular passages from the book as our test subject. NER will then be applied to these selected passages, allowing us to measure the important metrics of accuracy and F-value, which demonstrate the system's ability to identify entities in a variety of contexts and text types.



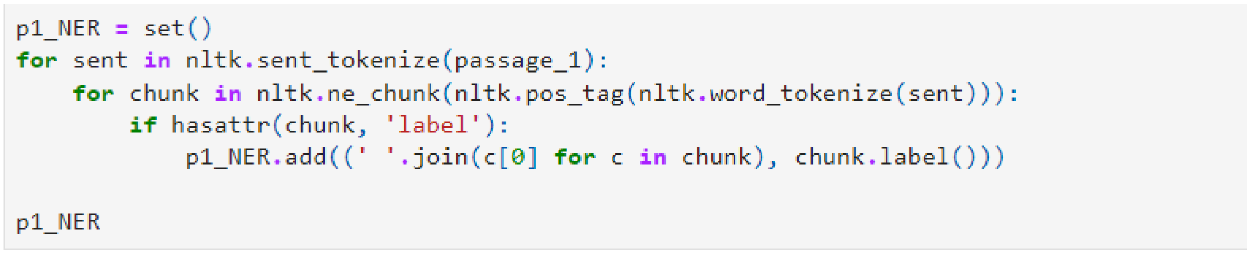
```
passage_1 = "LANGDON FELT FIRM hands lifting him now … urging him from his delirium, helping him out ofthe taxi. The pavement felt cold beneath h

passage_2 = "I'M IN FLORENCE!? Robert Langdon's head throbbed. He was now seated upright in his hospital bed,repeatedly jamming his finger into t

passage_3 = "Snaking through heavy crowds on the Riva degli Schiavoni, Langdon, Sienna, and Ferris hugged the water's edge, making their way into
```

Passage 1



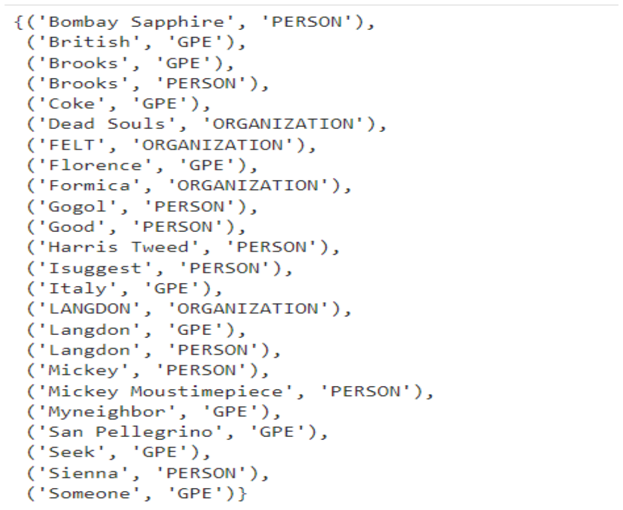
```
passage_1 = "LANGDON FELT FIRM hands lifting him now … urging him from his delirium, helping him out ofthe taxi. The pavement felt cold beneath h
```

➢ NER



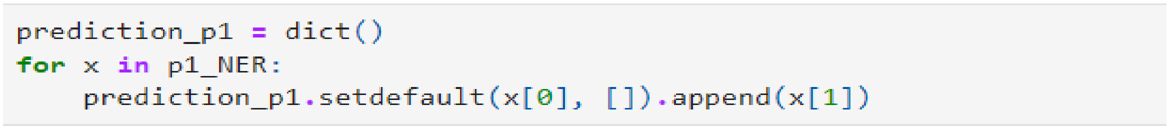
```
p1_NER = set()
for sent in nltk.sent_tokenize(passage_1):
    for chunk in nltk.ne_chunk(nltk.pos_tag(nltk.word_tokenize(sent))):
        if hasattr(chunk, 'label'):
            p1_NER.add((' '.join(c[0] for c in chunk), chunk.label()))

p1_NER
```

Output:



```
{('Bombay Sapphire', 'PERSON'),
 ('British', 'GPE'),
 ('Brooks', 'GPE'),
 ('Brooks', 'PERSON'),
 ('Coke', 'GPE'),
 ('Dead Souls', 'ORGANIZATION'),
 ('FELT', 'ORGANIZATION'),
 ('Florence', 'GPE'),
 ('Formica', 'ORGANIZATION'),
 ('Gogol', 'PERSON'),
 ('Good', 'PERSON'),
 ('Harris Tweed', 'PERSON'),
 ('Isuggest', 'PERSON'),
 ('Italy', 'GPE'),
 ('LANGDON', 'ORGANIZATION'),
 ('Langdon', 'GPE'),
 ('Langdon', 'PERSON'),
 ('Mickey', 'PERSON'),
 ('Mickey Moustimepiece', 'PERSON'),
 ('Myneighbor', 'GPE'),
 ('San Pellegrino', 'GPE'),
 ('Seek', 'GPE'),
 ('Sienna', 'PERSON'),
 ('Someone', 'GPE')}
```

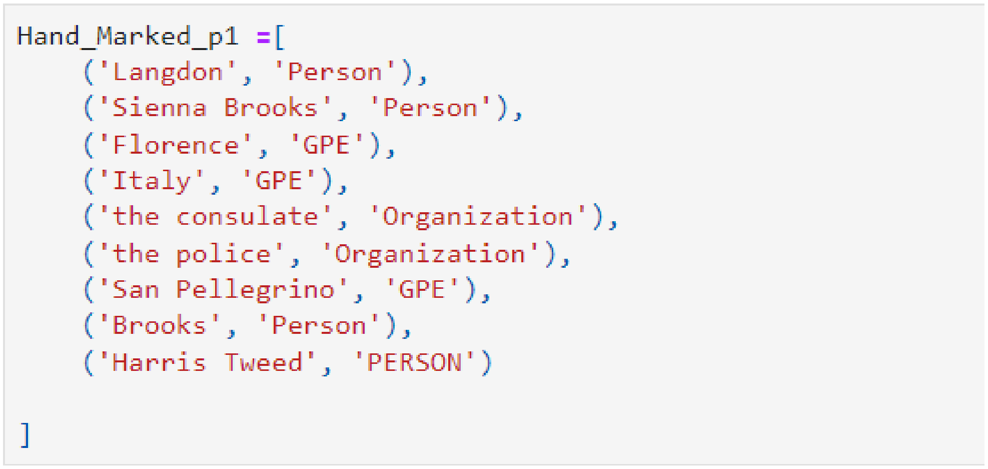
➢ Predictions



```
prediction_p1 = dict()
for x in p1_NER:
    prediction_p1.setdefault(x[0], []).append(x[1])
```

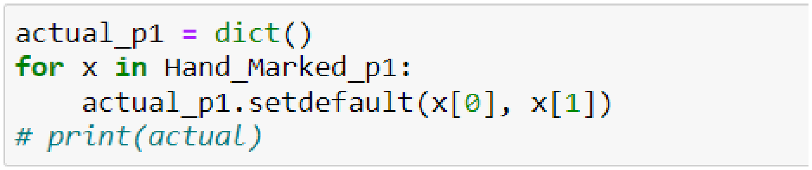
this code organizes the entities extracted from p1_NER into a dictionary (prediction_p1) where entities act as keys and their associated labels are stored as values in lists. This structure enables easy access and manipulation of entities and their labels for further analysis or processing.

➢ Manual Labeling



```
Hand_Marked_p1 =[
    ('Langdon', 'Person'),
    ('Sienna Brooks', 'Person'),
    ('Florence', 'GPE'),
    ('Italy', 'GPE'),
    ('the consulate', 'Organization'),
    ('the police', 'Organization'),
    ('San Pellegrino', 'GPE'),
    ('Brooks', 'Person'),
    ('Harris Tweed', 'PERSON')

]
```

➢ Actual



```
actual_p1 = dict()
for x in Hand_Marked_p1:
    actual_p1.setdefault(x[0], x[1])
# print(actual)
```

This code segment creates a dictionary named *actual_p1* by iterating through a list of tuples called *Hand_Marked_p1*. Each tuple in Hand_Marked_p1 seems to represent an entity and its associated label or category.

➢ Performance Measures

```
tp, tn, fp, fn = 0, 0, 0, 0
```

```
for entity, type_ in actual_p1.items():
    if entity in prediction_p1:
        if type_ in prediction_p1[entity]:
            tp += 1
            fp += len(prediction_p1[entity]) - 1
        else:
            fn += 1
    else:
        fn += 1
```

Here, tp= True Positive, tn = True Negative, fp = False Positive, fn = False Negative.

This code calculates performance metrics such as True Positives (tp), False Positives (fp), and False Negatives (fn) for evaluating the named entity recognition (NER) system. It compares the entities and their labels in actual_p1 (manually labeled) with those in prediction_p1 (predicted by the NER system).

Output:

```
print("TP: ", tp, " TN: ", tn, " FP: ", fp, " FN: ", fn)
```

```
TP:  4  TN:  0  FP:  0  FN:  5
```

➢ F-Values and Accuracy

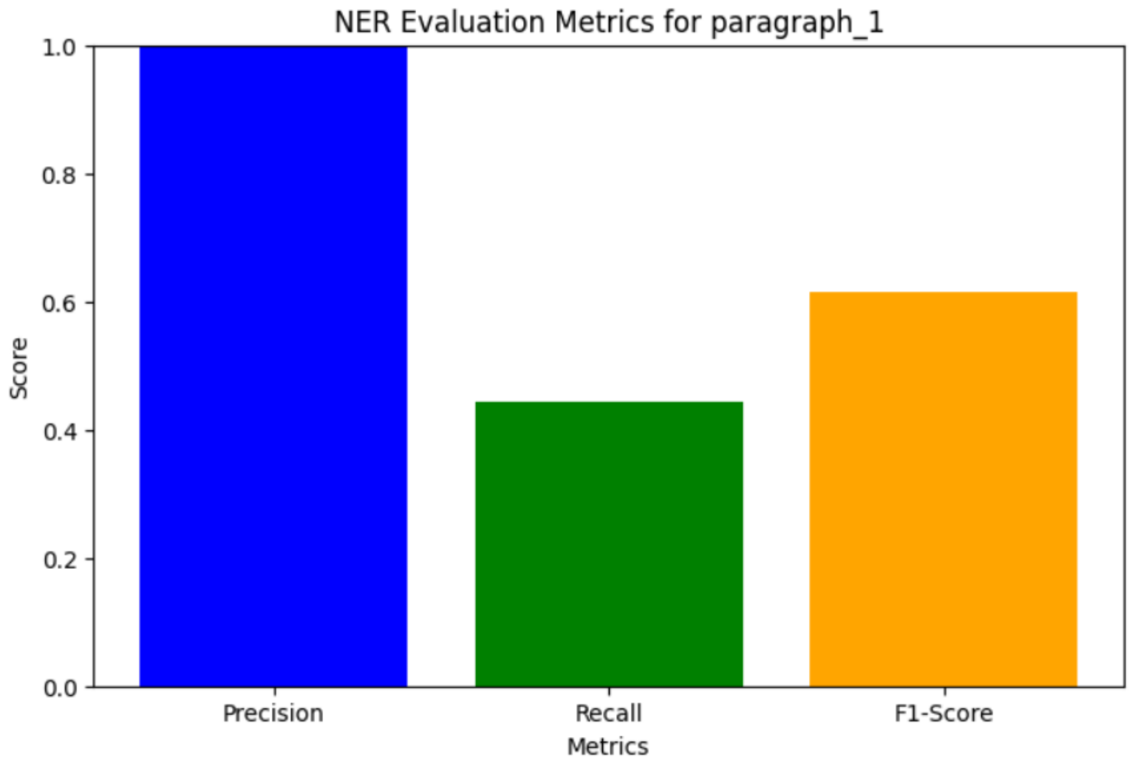
```
F1_score_p1 = (2 * tp) / ((2 * tp) + fp + fn)
accuracy = (tp + tn) / (tp + tn + fn + fp)
print("Accuracy: {0:.2f}".format(accuracy))
print("F1 Score: {0:.2f}".format(F1_score_p1))
```

```
Accuracy: 0.44
F1 Score: 0.62
```

F1 score is a measure that combines precision and recall into a single metric, providing a balance between the two. It's calculated using the formula:

F1 = 2* (Precision)*(recall)/(Precision + Recall)

We get an Accuracy of 44% and F1 score of 0.62 for this passage.



Similarly, we can find the accuracy and F1-values for the other two passages.

Passage 2

➢ Performance Measures

```
tp, tn, fp, fn = 0, 0, 0, 0
```

```
for entity, type_ in actual_p2.items():
    if entity in prediction_p2:
        if type_ in prediction_p2[entity]:
            tp += 1
            fp += len(prediction_p2[entity]) - 1
        else:
            fn += 1
    else:
        fn += 1
```

Output:

```
print("TP: ", tp, " TN: ", tn, " FP: ", fp, " FN: ", fn)
```

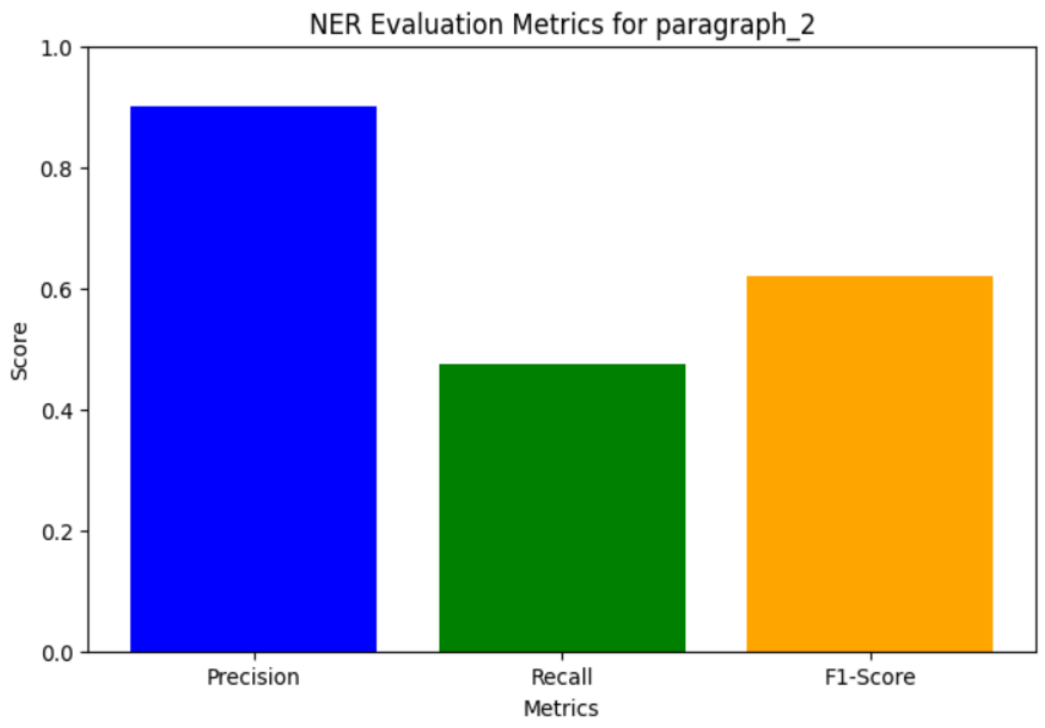
```
TP:  9  TN:  0  FP:  1  FN:  10
```

➢ F-Values and Accuracy

```
F1_score_p2 = (2 * tp) / ((2 * tp) + fp + fn)
accuracy = (tp + tn) / (tp + tn + fn + fp)
print("Accuracy: {0:.2f}".format(accuracy))
print("F1 Score: {0:.2f}".format(F1_score_p2))
```

```
Accuracy: 0.45
F1 Score: 0.62
```

We get an Accuracy of 45% and F1 measure of 0.62 from this passage.



Passage 3

➢ Performance Measures

```
tp, tn, fp, fn = 0, 0, 0, 0
```

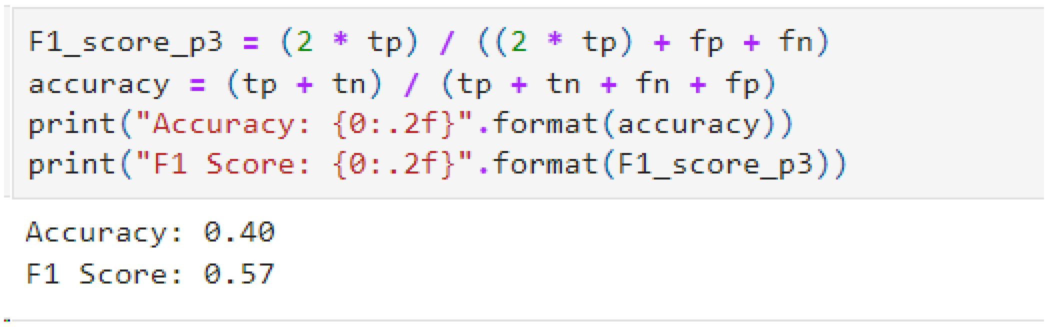
```
for entity, type_ in actual_p3.items():
    if entity in prediction_p3:
        if type_ in prediction_p3[entity]:
            tp += 1
            fp += len(prediction_p3[entity]) - 1
        else:
            fn += 1
    else:
        fn += 1
```

Output:

```
print("TP: ", tp, " TN: ", tn, " FP: ", fp, " FN: ", fn)
```

```
TP:  10  TN:  0  FP:  4  FN:  11
```

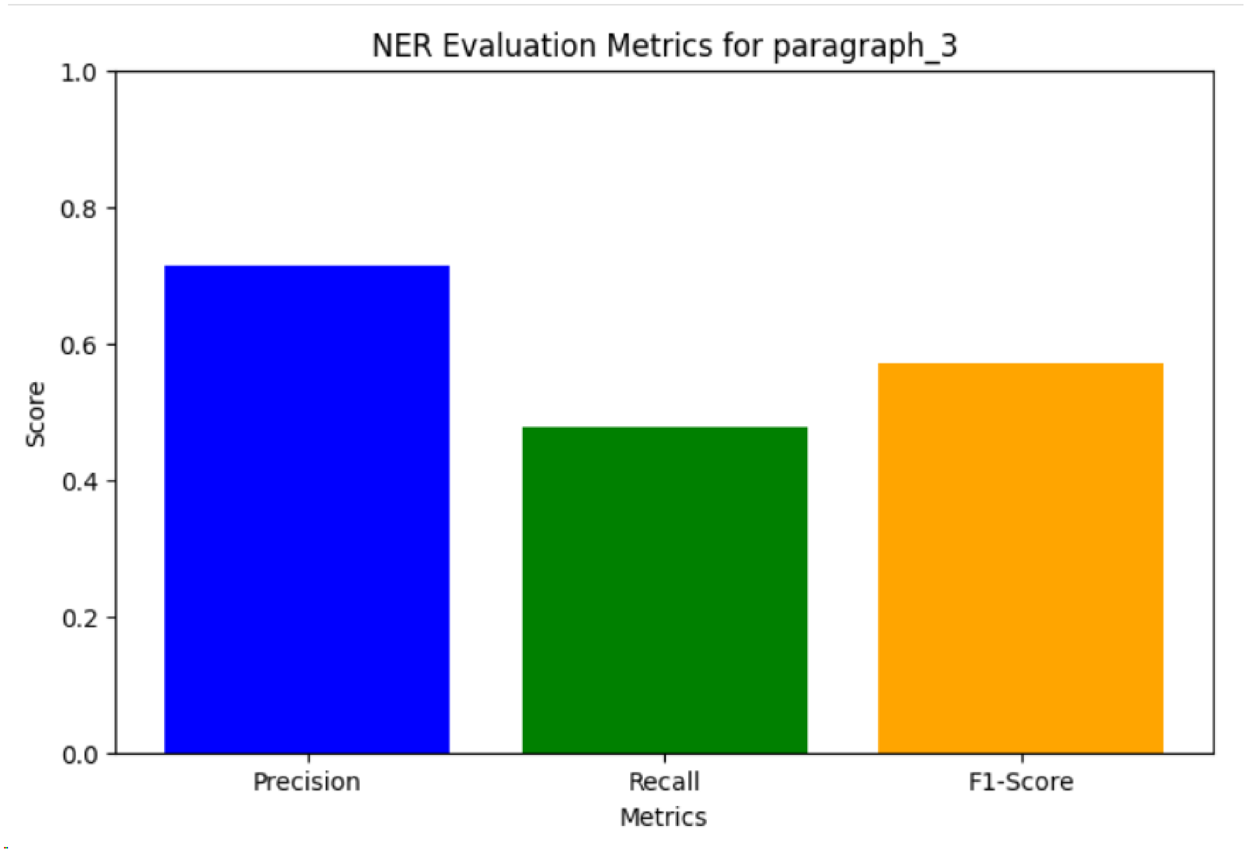
➢ F-Values and Accuracy



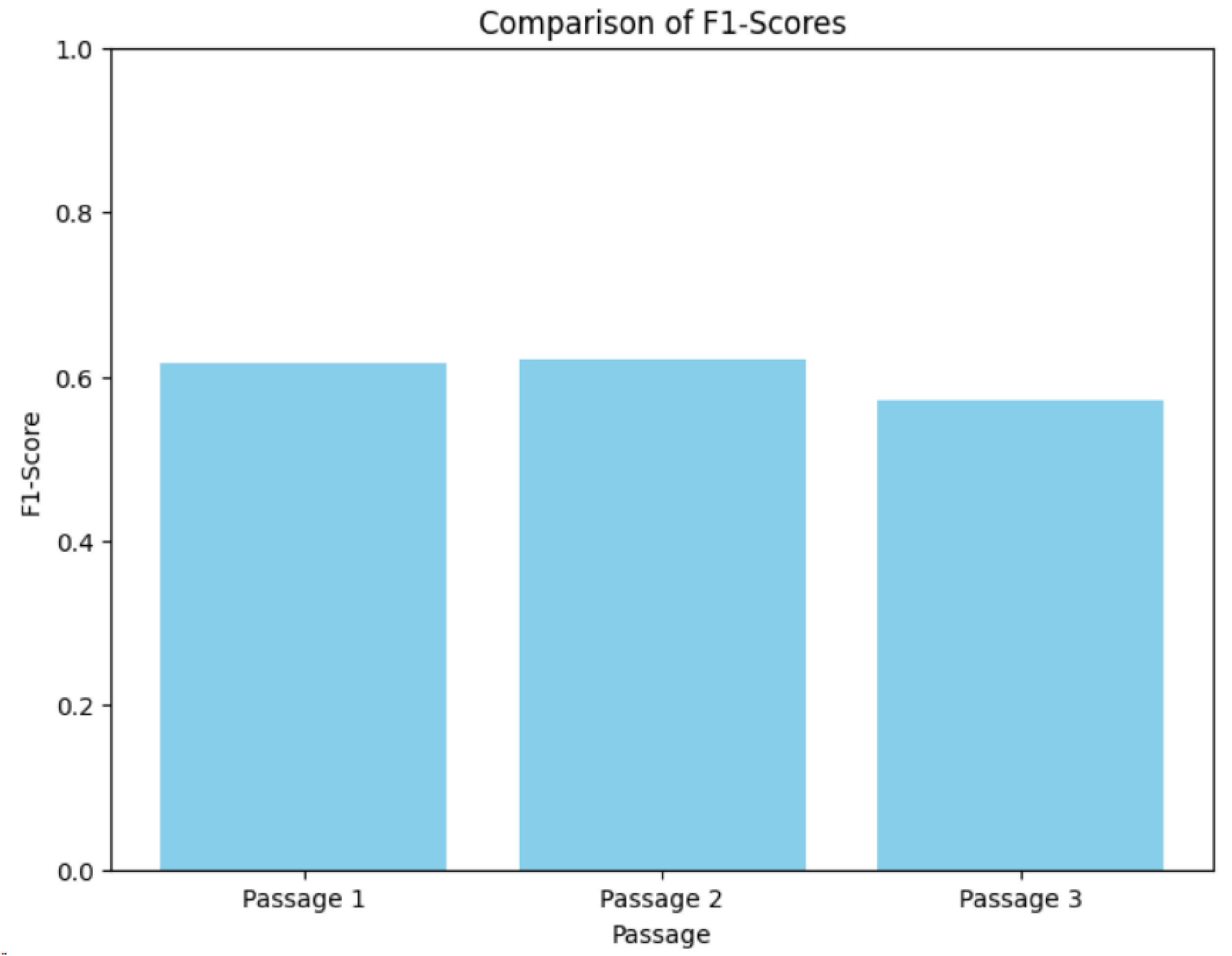
```
F1_score_p3 = (2 * tp) / ((2 * tp) + fp + fn)
accuracy = (tp + tn) / (tp + tn + fn + fp)
print("Accuracy: {0:.2f}".format(accuracy))
print("F1 Score: {0:.2f}".format(F1_score_p3))
```

```
Accuracy: 0.40
F1 Score: 0.57
```

We get an Accuracy of 40% and F1 measure of 0.57 from this passage.



F1 Scores Comparison


Comparison of F1-Scores
1.0
0.8
0.6
0.4
0.2
0.0
F1-Score
Passage 1
Passage 2
Passage 3
Passage

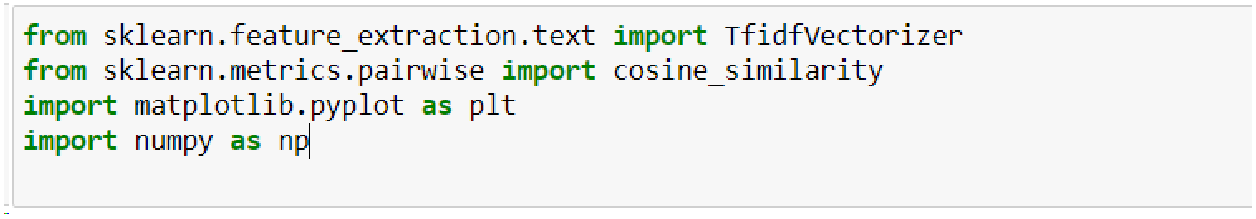
Part 2

Objective

The primary objective of this task is to generate Term Frequency-Inverse Document Frequency (TF-IDF) vectors for individual chapters and subsequently assess the similarity between these chapters using a relevant similarity measure. The TF-IDF vectors will capture the importance of terms within each chapter, allowing us to quantify the textual content. By employing a suitable similarity measure, we aim to identify and visualize the degree of similarity between pairs of chapters.

Importing Libraries

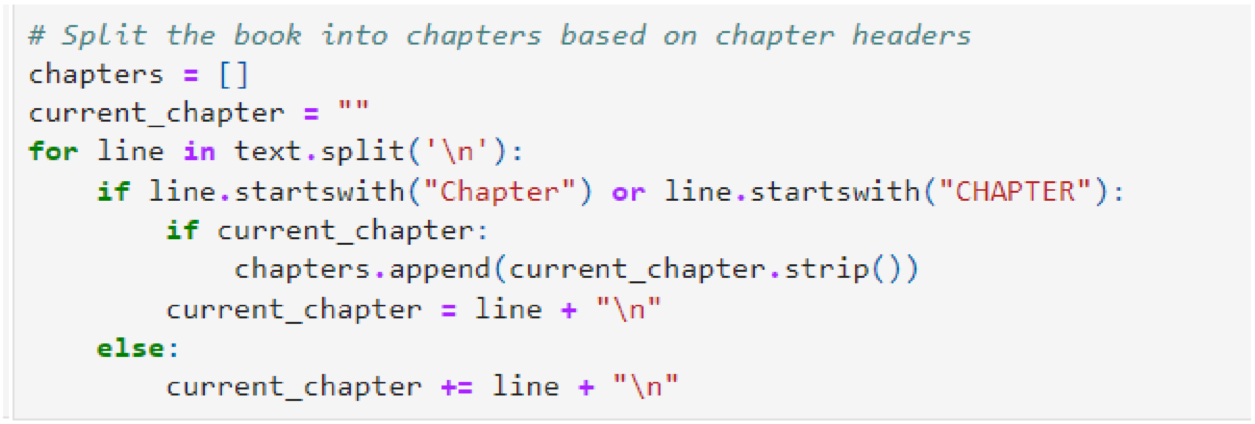
To fulfill our goals, we must import necessary libraries into the code and perform essential operations on our textbook.



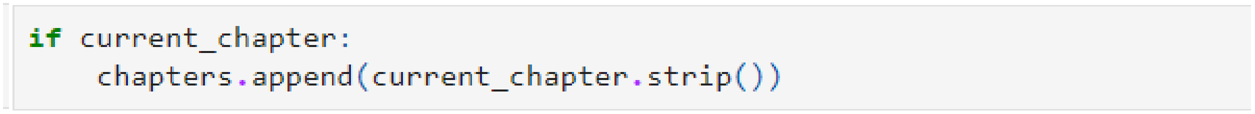
```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
import numpy as np
```

Split the book into chapters

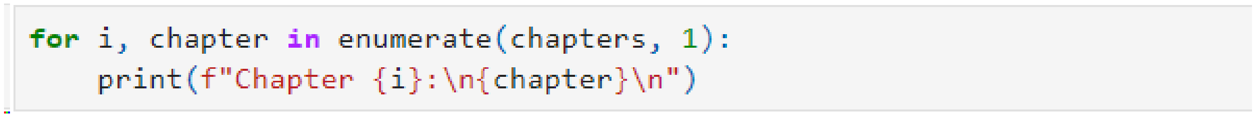
This code is designed to split a text document into chapters based on chapter headers. It iterates through the lines of the text and identifies chapter headers (lines starting with "Chapter" or "CHAPTER"). Upon encountering a header, it separates the text content into distinct chapters and stores them in a list named chapters.



```python
# Split the book into chapters based on chapter headers
chapters = []
current_chapter = ""
for line in text.split('\n'):
    if line.startswith("Chapter") or line.startswith("CHAPTER"):
        if current_chapter:
            chapters.append(current_chapter.strip())
        current_chapter = line + "\n"
    else:
        current_chapter += line + "\n"
```



```python
if current_chapter:
    chapters.append(current_chapter.strip())
```



```python
for i, chapter in enumerate(chapters, 1):
    print(f"Chapter {i}:\n{chapter}\n")
```

Here's a breakdown of the code's functionality:

- **Initialization and Iteration:**

The code initializes an empty list of chapters to store individual chapters and a string *current_chapter* to accumulate the text for each chapter.

It iterates through each line of the text using *text.split('\n')*.

- **Identifying Chapter Headers:**

For each line, it checks if the line starts with "Chapter" or "CHAPTER" using *startswith*.

If a line matches a chapter header, it indicates the start of a new chapter. It appends the accumulated *current_chapter* to the chapters list and starts a new current_chapter with the current line.

- **Assembling Chapters:**

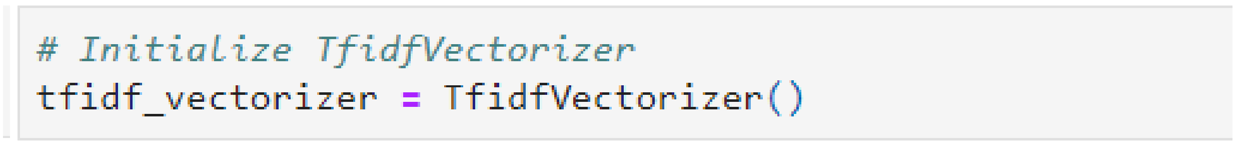
Non-header lines are appended to the current_chapter string, thereby accumulating the text content for each chapter.

- **Printing Chapters**:

Finally, it prints each chapter along with its corresponding number using an enumeration, indicating the chapter number and its content.

Initializing TF-IDF vector

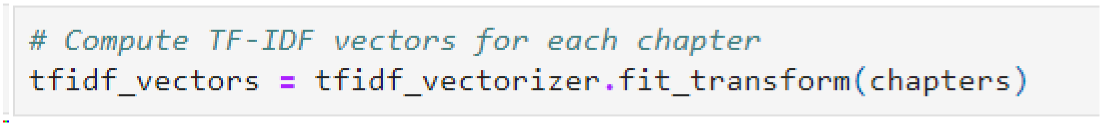
This vectorizer is a crucial component in converting text data into numerical representations using the TF-IDF (Term Frequency-Inverse Document Frequency) technique.



```
# Initialize TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer()
```

The *TfidfVectorizer* is a class used for converting a collection of raw text documents into a matrix of TF-IDF features.

TF-IDF vector for each chapters



```
# Compute TF-IDF vectors for each chapter
tfidf_vectors = tfidf_vectorizer.fit_transform(chapters)
```

Similarities between TF-IDF vectors

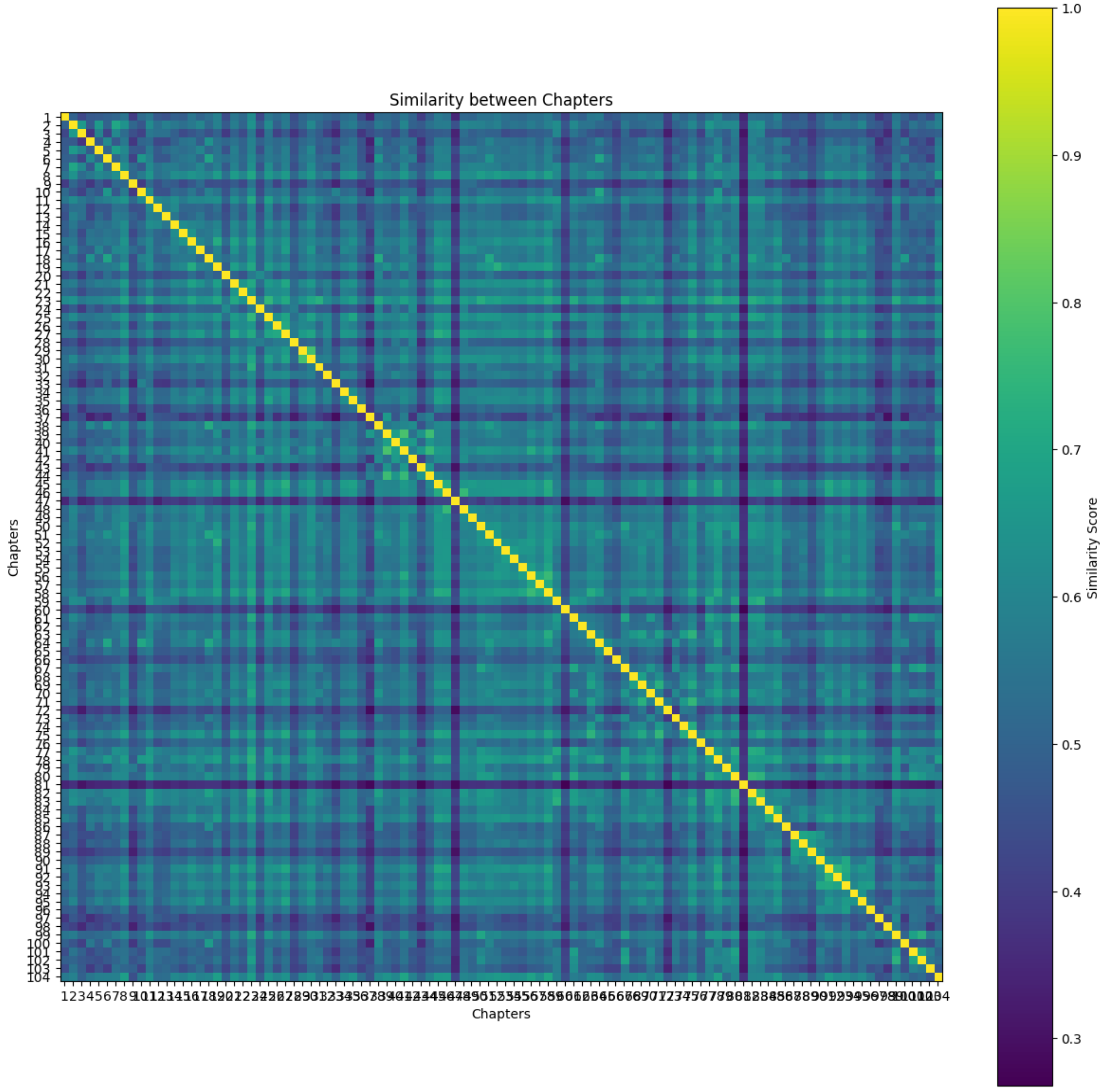
```
# Calculate cosine similarity between TF-IDF vectors
similarity_scores = cosine_similarity(tfidf_vectors)
```

We use cosine similarities to find similarities between TF-IDF vectors of chapters.

Gradient table

```
# Visualize similarity scores as a gradient table
plt.figure(figsize=(15, 15))
plt.imshow(similarity_scores, cmap='viridis', interpolation='nearest')
plt.colorbar(label='Similarity Score')
plt.title('Similarity between Chapters')
plt.xlabel('Chapters')
plt.ylabel('Chapters')
plt.xticks(np.arange(len(chapters)), np.arange(1, len(chapters) + 1))
plt.yticks(np.arange(len(chapters)), np.arange(1, len(chapters) + 1))
plt.savefig('similarity_table.png', bbox_inches='tight')
plt.show()
```

Output:


Similarity between Chapters
Chapters
Chapters
Similarity Score
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3