# Natural Language Processing

## Team - NLP_PROJECT_23


### Team Members

Dhruv Prabhugaonkar - 21ucs069

Dhruv Soni - 21ucs069

Harsh Kamboj - 21ucs085

Gautam Mittal - 21ucs081

Course Instructor
Dr. Sakthi Balan Muthiah
Department of Computer Science Engineering
The LNM Institute of Information Technology

# PROJECT ROUND 1

---

[Github Code Link]

## Book Referred

**The Hound of the Baskervilles** by Arthur Conan Doyle

## Project Overview

In this project, we will be analysing the novel **The Hound of the Baskervilles**. We will pre-process the novel, tokenize and apply POS Tagging to the novel. We will use Python libraries to accomplish our tasks.

# GOALS

1. Import Text in text format (call it as T)
2. Pre-processing of T
3. Tokenize the T
4. Remove Stop Words
5. Analyse the frequency distribution of tokens in T
6. Word Cloud of Tokens
7. POS tagging using Tagset (TreeBank here)
8. Get BiGram probability table for largest Chapter
9. Play Shanon game with another chapter using previous probability table

# Libraries used

1. Pandas - for data manipulation and analysis
2. NLTK - for tokenization , frequency distribution, stopwords
3. Re - to use regular expressions
4. Matplotlib.pyplot - for data visualisations
5. Word cloud - Used to create WordClouds from Tokenized Data
6. Seaborn - visualisation of frequency distribution

# Description Of Data in text file

```
The Hound of the
Baskervilles
By Arthur Conan Doyle

Download free eBooks of classic literature, books and
novels at Planet eBook. Subscribe to our free eBooks blog
and email newsletter.


Chapter 1
Mr. Sherlock Holmes


Mr. Sherlock Holmes, who was usually very late in the
mornings, save upon those not infrequent occasions
when he was up all night, was seated at the breakfast table.
I stood upon the hearth-rug and picked up the stick which
our visitor had left behind him the night before. It was a
fine, thick piece of wood, bulbous-headed, of the sort which
is known as a 'Penang lawyer.' Just under the head was a
broad silver band nearly an inch across. 'To James Mortimer, M.R.C.S., from his friends of the C.C.H.,' was engraved
upon it, with the date '1884.' It was just such a stick as the
old-fashioned family practitioner used to carry—dignified,
solid, and reassuring.
'Well, Watson, what do you make of it?'
Holmes was sitting with his back to me, and I had given
him no sign of my occupation.
'How did you know what I was doing? I believe you have
eyes in the back of your head.'
'I have, at least, a well-polished, silver-plated coffee-pot
in front of me,' said he. 'But, tell me, Watson, what do you
make of our visitor's stick? Since we have been so unfortu-
```

# Observations

Pre-processing requirements : We remove the title name, punctuations, chapter name, page numbers, running sections, empty spaces

Convert the text to lower case for a more flowing data to work with.

# TASKS

## 1. Import the book as file T.txt

We also remove the sentence that only occurs once in the starting of the downloaded book. We remove the title, white spaces, and page numbers.

```
#To open the file
file = open(r"T.txt",encoding='utf-8')
listofwords = file.read().splitlines()
page_number_pattern = re.compile(r'\d+')
listofwords = [i for i in listofwords if i!='' and i!='The Hound of the Baskervilles' and i!='\x18'
            and not page_number_pattern.match(i) and i!='Download free eBooks of classic literature, books and'
            and i!='novels at Planet eBook. Subscribe to our free eBooks blog' and i!='and email newsletter.']
text = " ".join(listofwords)
```

## 2. Text Preprocessing

Removing the punctuation marks, chapter number, running section (footer) and converting to lowercase.

```
#string which contains the punctuations which we want removed
punctuations = '''!()-[]{};:'"\,<>./‘’?“”@#$%^&*_~'''
processed_text = ""
for i in text:
    if i not in punctuations:
        processed_text = processed_text + i

#Making the result lowercase
processed_text = processed_text.lower()
substring_to_remove_1 = "free ebooks at planet ebookcom"
processed_text= processed_text.replace(substring_to_remove_1, "")
pattern = r'chapter \d+'
processed_text = re.sub(pattern, '', processed_text)
print(processed_text[:100])#print for 100 characters
```

```
the hound of the baskervilles by arthur conan doyle  mr sherlock holmes mr sherlock holmes who was u
```

### 3. Tokenizing T

Certainly, here are the sentences with numbering:

1. Next, we tokenize the processed text using the 'word_tokenize' function from the nltk.tokenize library.

```
tokenisedtext = word_tokenize(processed_text)
print(tokenisedtext[:10])
```

```
['the', 'hound', 'of', 'the', 'baskervilles', 'by', 'arthur', 'conan', 'doyle', 'mr']
```

2. Tokenizers break down strings into lists of substrings.

3. The 'word_tokenize' tokenizer requires the installation of the Punkt sentence tokenization model.

### 4. Remove Stopwords from T

Stopwords are common, non-significant words like "the," "and," "in" that are often removed from text data during NLP tasks to reduce noise and improve efficiency.

```
# Stop words need to be removed.
stop_words = set(stopwords.words('english'))
final_tokens = [i for i in tokenisedtext if not i in stop_words]
finaltext = "  "
finaltext = finaltext.join(final_tokens)
print(finaltext[:100]) #after removing stopwords
```

```
hound  baskervilles  arthur  conan  doyle  mr  sherlock  holmes  mr  sherlock  holmes  usually  late
```

## 5. Analyse the frequency distribution of tokens in T

We use the nltk.FreqDist() function from the **nltk** library to calculate the .frequency of tokens

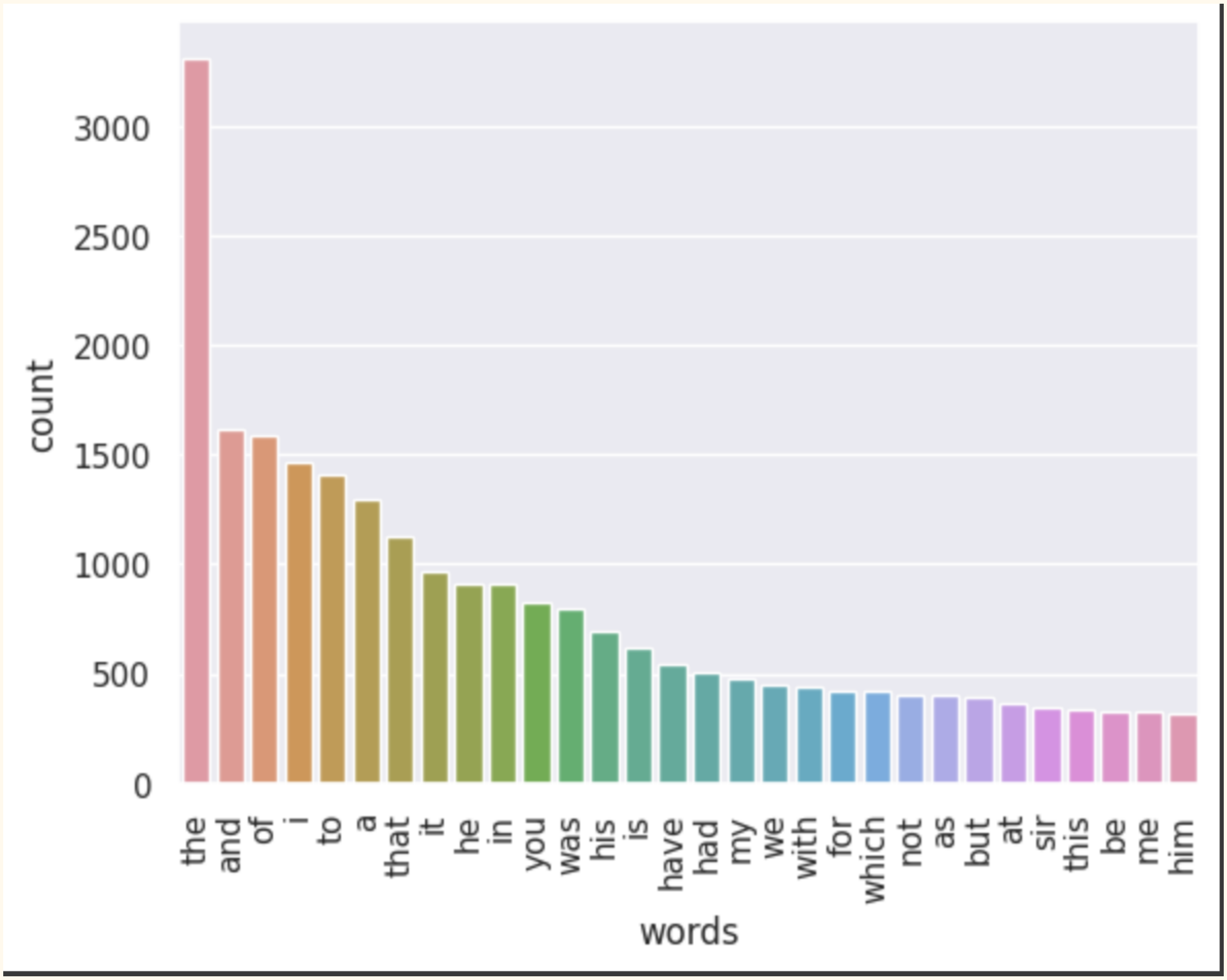
```
#Frequency Distrubution
freq_dist=nltk.FreqDist(finaltext)
print(freq_dist.most_common(15))
freq_dist=list(freq_dist)
```

```
[(' ', 53174), ('e', 19612), ('r', 11212), ('a', 11147), ('n', 11044), ('s', 10998), ('o', 10779),
```

## 6. Visualise the frequency distribution of 30 most occurring tokens in T

```
import seaborn as sb
sb.set(style='darkgrid')
dataf=pd.DataFrame(tokenisedtext)
```

```
sb.countplot(x=dataf[0],order=dataf[0].value_counts().iloc[:30].index)
plt.xticks(rotation=90)
plt.xlabel('words')
plt.show()
#Plotting the counts of the most frequent words ordered in descending order of their frequencies.
```




### 7. Create a word cloud

A word cloud is a visual representation of text data where words are displayed in varying sizes, with the most frequently occurring words appearing larger and less frequent words appearing smaller.

```
# Word cloud
wc = WordCloud(width = 800, height = 600,
                background_color ='white',
                min_font_size = 10,stopwords = {},colormap='winter').generate(finaltext)

plt.figure(figsize = (12,8), facecolor = None)
plt.imshow(wc)
plt.axis("off")
plt.tight_layout(pad = 0)
plt.show()
```

## World Cloud

## 8. POS Tagging using the treebank tagset

```
nltk.download("treebank")
pos_tags = nltk.pos_tag(final_tokens)
print(pos_tags[:20])
```

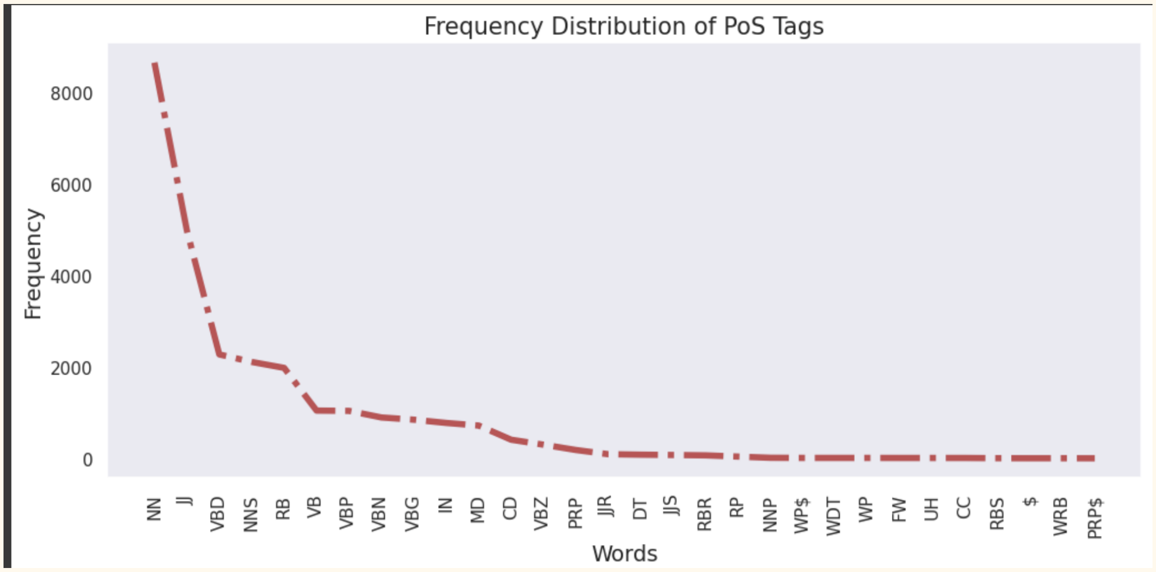
```
[nltk_data] Downloading package treebank to /root/nltk_data...
[nltk_data]   Unzipping corpora/treebank.zip.
[('hound', 'NN'), ('baskervilles', 'NNS'), ('arthur', 'VBP'), ('conan', 'JJ'), ('doyle', 'JJ'), ('download', 'NN'), ('fre
```

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

```
Counter({'NN': 8645, 'JJ': 4981, 'VBD': 2270, 'NNS': 2106, 'RB': 1976, 'VB': 1043, 'VBP': 1037, 'VBN': 893, 'VBG': 844, '
```

## 9. Frequency Distribution of Various Tags

```
#Frequency Distribution of varios PoS Tags

pos_tags_freq = nltk.FreqDist(counts)
pos_tags_freq = {k: v for k, v in sorted(pos_tags_freq.items(), key=lambda item: item[1],reverse=True)}
x = list(pos_tags_freq.keys())[:40]
y = list(pos_tags_freq.values())[:40]
plt.figure(figsize=(12,5))
plt.plot(x,y,c='r',lw=4,ls='-.')
plt.grid()
plt.xticks(rotation=90)
plt.title('Frequency Distribution of PoS Tags',size=15)
plt.xlabel('Words',size=14)
plt.ylabel('Frequency',size=14)
plt.show()
```



## 10. Bi-Gram Probability Table for largest chapter C

We first create bigram tuples from the chapter C, then created a bigram probability table and displayed a 5*5 matrix for illustration and verification of the table creation.

```
# Generate bigrams
bigram_tuples = list(bigrams(tokens))

# Count bigram frequencies
bigram_freq = FreqDist(bigram_tuples)

# Create a bigram probability table
bigram_probabilities = {}

for bigram in bigram_freq:
    preceding_word, following_word = bigram
    if preceding_word not in bigram_probabilities:
        bigram_probabilities[preceding_word] = {}

    probability = bigram_freq[bigram] / tokens.count(preceding_word)
    bigram_probabilities[preceding_word][following_word] = probability
```

```
# Create a counter for preceding words
preceding_word_count = 0
#bigram probability table displayed for 5x5
for preceding_word in bigram_probabilities:
    if preceding_word_count < 5:

        following_word_count = 0
        for following_word, probability in bigram_probabilities[preceding_word].items():
            if following_word_count < 5:
                print(f"{preceding_word}, {following_word}, Probability : {probability:.4f}")
                following_word_count += 1
        preceding_word_count += 1
```

```
of, the, Probability : 0.2708
of, his, Probability : 0.0579
of, a, Probability : 0.0535
of, it, Probability : 0.0315
of, my, Probability : 0.0252
in, the, Probability : 0.2750
in, a, Probability : 0.0759
in, his, Probability : 0.0649
in, my, Probability : 0.0385
in, this, Probability : 0.0330
it, was, Probability : 0.1746
it, is, Probability : 0.1601
it, i, Probability : 0.0269
it, and, Probability : 0.0258
it, would, Probability : 0.0217
i, have, Probability : 0.0997
i, had, Probability : 0.0689
i, am, Probability : 0.0546
i, was, Probability : 0.0539
i, could, Probability : 0.0505
the, moor, Probability : 0.0420
the, man, Probability : 0.0145
the, same, Probability : 0.0112
the, baronet, Probability : 0.0112
the, matter, Probability : 0.0106
```

## 11. Playing the Shannon Game

In this step , we took some sentences from a random chapter in the book different from C . We removed the last word of the sentences . Then we used the bigram probability table obtained in the previous step to predict the last word . We found that accuracy in predicting the exact same word was 40%.

```python
correct_predictions = 0

for sentence, expected_next_word in sentences:
    words = sentence.lower().split()  # Tokenize the given sentence

    current_word = words[-1]  # Take the last word as the current word

    if current_word in bigram_probabilities:
        next_word = random.choices(
            list(bigram_probabilities[current_word].keys()),
            weights=list(bigram_probabilities[current_word].values())
        )[0]
    else:
        # If the current word doesn't have associated bi-grams, choose a random word
        next_word = random.choice(list(bigram_probabilities.keys()))

    if next_word == expected_next_word:
        correct_predictions += 1

    generated_sentence = ' '.join(words + [next_word])
    print(f"Generated: {next_word}\nExpected: {expected_next_word}\n")

accuracy = (correct_predictions / len(sentences)) * 100
print(f"Accuracy: {accuracy:.2f}%")
```

```
Generated: hours
Expected: hours

Generated: first
Expected: purpose

Generated: the
Expected: he

Generated: public
Expected: Barrymore’s

Generated: could
Expected: have

Generated: position
Expected: position

Generated: would
Expected: would

Generated: of
Expected: of

Generated: bogie
Expected: butterfly-net

Generated: man
Expected: interruption

Accuracy: 40.00%
```

## Inference

Applied POS tagging for all the tokens present in the book

Analysed the frequency distribution of the tokens

The most frequent Part of speech in the book was NOUN (NN) with 8645 count

Applied the Bigram probability table to predict the last word of a sentence from the Test set and found its accuracy as 40% correct.

## Conclusion

In this round of the project , we learnt how to preprocess the data, word tokenization , concept of stop words , concept of word cloud and its generation , POS tagging of the tokens , obtained the bigram distribution and played the Shannon game.

# PROJECT ROUND 2

---

## GOALS

### First Part:

(1) First recognise all the entities and then

(2) recognise all entity types.

(3) Use performance measures to measure the performance of the method used – For evaluation you take a considerable amount of random passages from the book, do a manual labelling and then compare your result with it. Repeat this evaluation three times and list out the F1 scores for all.

(4) Give a summary of the results through a good visualisation wherever necessary.

### Second Part:

1. Generate TF-IDF vectors for all the chapters separately and check which chapters are more similar by using a similarity measure. Visualise this as a gradient table with the score on it.

# Libraries used

1. Pandas - for data manipulation and analysis
2. NLTK - for tokenization , frequency distribution, stopwords
3. Spacy - to perform entity recognition in text
4. Os - to handle file operations
5. Numpy and matplotlib.pyplot - for visualising similarity scores
6. Sklearn - for TF-IDF vectorization and similarity calculations

# First part: Named Entity Recognition

### Recognizing all the entities -

Using the spacy library , we first recognized all the entities that are present in the book.

```
import spacy

# Load the English core web model
nlp = spacy.load("en_core_web_sm")

# Define a function to recognize entities and their types in a text
def recognize_entities(text):
    doc = nlp(text)
    entities = [(ent.text, ent.label_) for ent in doc.ents]
    return entities

# Read the book text from a file
with open("Preprocessed_text.txt", "r") as f:
    book_text = f.read()

# Recognize entities and their types in the book text
book_entities = recognize_entities(book_text)
print(len(book_entities))
```

```
744
```

Total entities found are 744.

First 20 entities for example are-

```
book_entities = recognize_entities(book_text)
for entity in book_entities[:20]:
  print(entity[0])
```

```
sherlock
sherlock
morning
night
penang
m.r.c.s.
polish
mortimer
watson
charing cross hospital inference
mortimer
house surgeon house
year ago
thirty
james m.r.c.s
devon house
jackson prize
swedish
london
hour
```

## Recognizing entity types:

In the Spacy library the entity types are as described below

| TYPE | DESCRIPTION |
|---|---|
| PERSON | People, including fictional |
| NORP | Nationalities or religious or political groups |
| FACILITY | Buildings, airports, highways, bridges, etc |
| ORG | Companies, agencies, institutions, etc |
| GPE | Countries, cities, states |
| LOC | Non-GPE locations, mountain ranges, bodies of water |
| PRODUCT | Objects, vehicles, foods, etc (Not services) |
| EVENT | Named hurricanes, battles, wars, sports events, etc |
| WORK_OF_ART | Titles of books, songs, etc |
| LAW | Named documents made into laws |
| LANGUAGE | Any named language |
| DATE | Absolute or relative dates or periods. |
| TIME | Times smaller than a day |
| PERCENT | Percentage, including "%". |
| MONEY | Monetary values, including unit |
| QUANTITY | Measurements, as of weight or distance |
| ORDINAL | "first", "second", etc |
| CARDINAL | Numerals that do not fall under another type |

After applying it to our book , following is the named entities along with their types

```
for entity, entity_type in book_entities:
        print(f"Entity: {entity}, Type: {entity_type}")
```

```
Entity: sherlock, Type: PERSON
Entity: sherlock, Type: PERSON
Entity: morning, Type: TIME
Entity: night, Type: TIME
Entity: penang, Type: GPE
Entity: m.r.c.s., Type: GPE
Entity: polish, Type: NORP
Entity: mortimer, Type: PERSON
Entity: watson, Type: PERSON
Entity: charing cross hospital inference, Type: ORG
Entity: mortimer, Type: PERSON
Entity: house surgeon house, Type: ORG
Entity: year ago, Type: DATE
Entity: thirty, Type: CARDINAL
Entity: james m.r.c.s, Type: PERSON
Entity: devon house, Type: ORG
Entity: jackson prize, Type: ORG
Entity: swedish, Type: NORP
Entity: london, Type: GPE
Entity: hour, Type: TIME
Entity: beg watson, Type: ORG
Entity: james
 , Type: PERSON
Entity: mortimer, Type: PERSON
Entity: marry, Type: PERSON
Entity: james mortimer, Type: PERSON
Entity: m.r.c.s, Type: GPE
Entity: holmes, Type: PERSON
Entity: sherlock, Type: PERSON
Entity: watson, Type: PERSON
Entity: holmes, Type: PERSON
Entity: holmes, Type: PERSON
Entity: second, Type: ORDINAL
```

## Performance Evaluation -

For performance evaluation , we took a random passage from the book . To avoid the overhead of preprocessing this again , we took it from the preprocessed file itself.

The first random passage and its hand labelled entities -

```
sample_text1 - "like moorland uproar call pistol horse flask wine length sense come craze mind thirteen number take horse start pursuit moon shine clear ride swiftly abreast take course maid need take reach home go mile pass night shepherd " ** NER **

handlabelled entity1= [("Hugo Baskerville", "PERSON"), ("Moorland", "LOC"), ("Moon", "ART"), ("Horse", "ANIMAL"), ("Flask", "PRODUCT"), ("Wine", "DRINK"), ("Thirteen", "CARDINAL"), ("Mile", "QUANTITY"), ("Shepherd", "PERSON"), ]
```

Now to compare the labelled vs the predicted entities , and to calculate the F1 score , following method was used-

```python
def calculate_f1_scores(sample_text, true_entities):
    # Load spaCy model
    nlp = spacy.load("en_core_web_sm")

    # Perform named entity recognition
    doc = nlp(sample_text)
    recognized_entities = [(ent.text.lower(), ent.label_) for ent in doc.ents]

    # Extract recognized entities of specified types
    valid_entity_types = ["LOC", "ART", "ANIMAL", "PRODUCT", "DRINK", "CARDINAL", "QUANTITY", "PERSON"]
    filtered_recognized_entities = [(text, label) for text, label in recognized_entities if label in valid_entity_types]

    # Generate true labels and recognized labels
    true_labels = [label for _, label in true_entities]
    recognized_labels = [label for _, label in filtered_recognized_entities]

    # Make sure the lengths are the same
    recognized_labels = recognized_labels + ['O'] * (len(true_labels) - len(recognized_labels))

    # Calculate entity-wise F1 score
    entity_f1_scores = classification_report(true_labels, recognized_labels, labels=valid_entity_types)

    # Calculate overall F1 score
    overall_f1_score = f1_score(true_labels, recognized_labels, labels=valid_entity_types, average='weighted')

    return entity_f1_scores, overall_f1_score
```

This function prints the precision , recall , f1 score and support of each entity and also prints the overall f1 score of the passage.

The process was repeated three times for different passages and the following is the result .

**Iteration 1 -**

Entity-wise F1 Score Sample 1:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| LOC | 0.00 | 0.00 | 0.00 | 1.0 |
| ART | 0.00 | 0.00 | 0.00 | 1.0 |
| ANIMAL | 0.00 | 0.00 | 0.00 | 1.0 |
| PRODUCT | 0.00 | 0.00 | 0.00 | 1.0 |
| DRINK | 0.00 | 0.00 | 0.00 | 1.0 |
| CARDINAL | 0.00 | 0.00 | 0.00 | 1.0 |
| QUANTITY | 0.00 | 0.00 | 0.00 | 1.0 |
| PERSON | 0.00 | 0.00 | 0.00 | 2.0 |
| micro avg | 0.00 | 0.00 | 0.00 | 9.0 |
| macro avg | 0.00 | 0.00 | 0.00 | 9.0 |
| weighted avg | 0.00 | 0.00 | 0.00 | 9.0 |

Overall F1 Score Sample 1: 0.0

**Iteration 2 -**

Entity-wise F1 Score Sample 2:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| LOC | 0.00 | 0.00 | 0.00 | 1 |
| ART | 0.00 | 0.00 | 0.00 | 0 |
| ANIMAL | 0.00 | 0.00 | 0.00 | 0 |
| PRODUCT | 0.00 | 0.00 | 0.00 | 0 |
| DRINK | 0.00 | 0.00 | 0.00 | 0 |
| CARDINAL | 0.00 | 0.00 | 0.00 | 0 |
| QUANTITY | 0.00 | 0.00 | 0.00 | 0 |
| PERSON | 0.50 | 0.33 | 0.40 | 3 |
| micro avg | 0.33 | 0.25 | 0.29 | 4 |
| macro avg | 0.06 | 0.04 | 0.05 | 4 |
| weighted avg | 0.38 | 0.25 | 0.30 | 4 |

Overall F1 Score Sample 2: 0.30000000000000004

## Iteration 3-

```
Entity-wise F1 Score Sample 3:
              precision    recall  f1-score   support

         LOC       1.00      0.33      0.50         3
         ART       0.00      0.00      0.00         0
      ANIMAL       0.00      0.00      0.00         0
     PRODUCT       0.00      0.00      0.00         0
       DRINK       0.00      0.00      0.00         0
    CARDINAL       0.00      0.00      0.00         0
    QUANTITY       0.00      0.00      0.00         0
      PERSON       0.00      0.00      0.00         1

   micro avg       1.00      0.25      0.40         4
   macro avg       0.12      0.04      0.06         4
weighted avg       0.75      0.25      0.38         4


Overall F1 Score Sample 3: 0.375
```

# Second part: TF-IDF vectors

## Generating TF-IDF vectors for all chapters

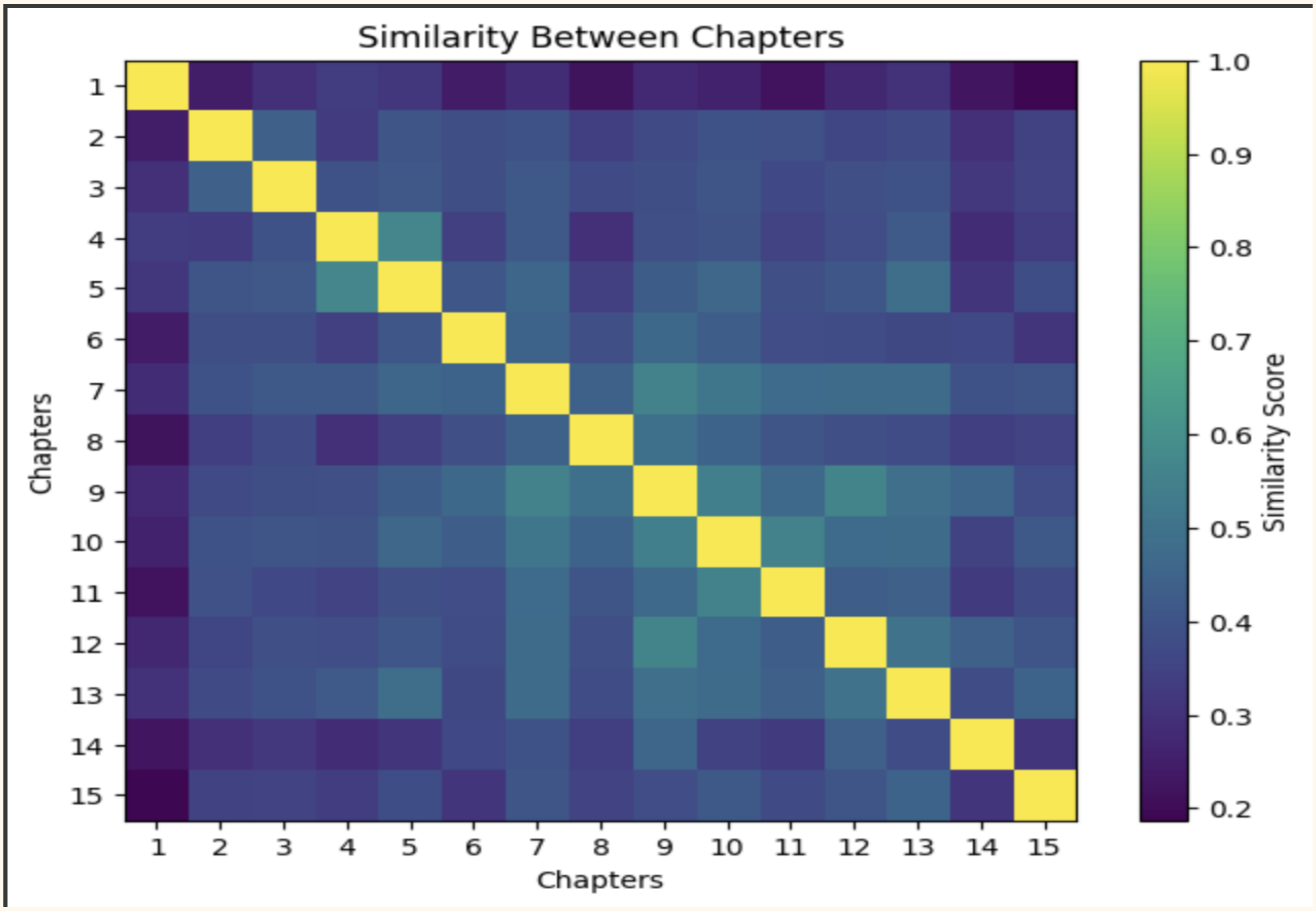
- Initialization a TF-IDF vectorization from Scikit-learn
- Use the fit_transform method to compute TF-IDF vectors for each preprocessed chapter. This step converts text into numerical vectors that represent the importance of words in each chapter.

```python
# Directory containing the preprocessed chapter text files
directory = "chapters"

# Read preprocessed chapters from separate text files
chapters = []
for i in range(1, 16):  # Assuming 15 chapters
    filename = os.path.join(directory, f"c{i}.txt")
    with open(filename, 'r', encoding='utf-8') as file:
        chapter_text = file.read()
        chapters.append(chapter_text)

# Initialize TF-IDF Vectorizer
tfidf_vectorizer = TfidfVectorizer()

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

## Calculate and Visualise the Similarity measure

- Calculate the cosine-similarity between each pair of chapters (total 15) and create a similarity score matrix.
- Create a gradient table representing similarity scores.

```
# Calculate cosine similarity between chapters
similarity_matrix = cosine_similarity(tfidf_matrix, tfidf_matrix)

# Visualize similarity scores in a gradient table
plt.figure(figsize=(8, 6))
plt.imshow(similarity_matrix, cmap='viridis', interpolation='nearest')
plt.title('Similarity Between Chapters')
plt.colorbar(label='Similarity Score')
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.xlabel('Chapters')
plt.ylabel('Chapters')
plt.show()
```


Similarity Between Chapters
Chapters
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Similarity Score
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2