February 14, 2024

Department of Computer Science and Engineering
Islamic University of Technology (IUT)
A subsidiary organ of OIC

Natural Language Processing, Assignment 1

Azmayen Fayek Sabil, 190042122

Contents

1 Task 1

1.1 Regular expressions

Regular expressions is basically used in text processing. We can use regular expression to find, extract patterns from a pile of text. Mainly it is used for searching, matching, and manipulating textual data. Here are some common regular expressions used in NLP:

1.2 Basic Text Cleansing

When working with downloaded books, it's common to perform basic text cleansing to ensure consistency and ease of analysis. Here's an example using Python's 're' module:

```
import re

def basic_text_cleansing(text):
    # Replacing various forms of whitespace with a single space
    text = re.sub('\s+', ' ', text)

    # Converting the text to lowercase
    text = text.lower()

    # Replacing common abbreviations with their full form
    text = re.sub("it's", 'it is', text)
    text = re.sub("don't", 'do not', text)
    text = re.sub("Don't", 'Do not', text)
    text = re.sub("I'm", 'I am', text)
#     text = re.sub("you are", 'you\'re', text)

    return text

cleansed_mySite_text = basic_text_cleansing(mySite.text)
```

In this example, the *basic_text_cleansing*(*text*) function takes the text of the downloaded book and performs the following operations using 're.sub':

1. Replaces various forms of whitespace with a single space using the regular expression `\s+`.
2. Converts all text to lowercase using 'lower()' method.
3. Replaces the abbreviation like "it's" with its full form "it is".

We can also extend the function by adding more 're.sub' calls for other common abbreviations or patterns that need to be replaced in your specific text data.

1.3 Sentence Extractor

We can use regular expression to extract sentences that matches a certain expression.
This Python code below demonstrates the implementation of a regular expression to extract complete sentences containing matches for the term `(P|p)hilo(?!nous)[a-z]+`. The code processes three philosophical works: "Beyond Good and Evil," "The Problems of Philosophy," and a self-chosen philosophy work which is "Man and Superman: A Comedy and a Philosophy".

```
import requests
import re

# URLs for the philosophical works
website1 = 'https://www.gutenberg.org/cache/epub/4363/pg4363.txt'
website2 = 'http://www.gutenberg.org/cache/epub/5827/pg5827.txt'
myWebsite = 'https://www.gutenberg.org/cache/epub/3328/pg3328.txt'

```

```
# Retrieve the text content of each work
site1 = requests.get(website1)
site2 = requests.get(website2)
mySite = requests.get(myWebsite)

# Regular expression for matching the term
search_regex = r'(P|p)hilo(?!nous)[a-z]+'

# Function to extract matching sentences
def get_matching_sentences(text, regex):
    results = [{"indices": m.span(), "match": m.group()} for m in re.finditer
   (regex, text)]
    matches = []
    for result in results:
        start, end = result["indices"]
        sentence_start = text.rfind(".", 0, start) + 1
        sentence_end = text.find(".", end)
        sentence = text[sentence_start:sentence_end].strip()
        matches.append({"match": result["match"], "sentence": sentence})
    return matches

# Apply the regular expression to each philosophical work
matches1 = get_matching_sentences(site1.text, search_regex)
matches2 = get_matching_sentences(site2.text, search_regex)
matches3 = get_matching_sentences(mySite.text, search_regex)
```

```
    OUTPUT:

    Random Subset of Matches in 'The Problems of Philosophy':
    Match: philosophy,
    Sentence: Such beliefs philosophy will bid us
    reject, unless some new line of argument is found to support them

    Match: Philosophy,
    Sentence: Philosophy
    may claim justly that it diminishes the risk of error, and that in some
    cases it renders the risk so small as to be practically negligible
```

1.3.1 Explanation of the Regular Expression

The regular expression `(P|p)hilo(?!nous)[a-z]+` breaks down as follows:

- `(P|p)`: Capturing group for either uppercase "P" or lowercase "p".
- `hilo`: Literal match for the characters "hilo".
- `(?!nous)`: Negative lookahead assertion excluding matches followed by "nous".
- `[a-z]+`: Matches one or more lowercase letters.

This expression captures instances of "Philo" or "philo" followed by at least one lowercase letter but excludes matches where "nous" immediately follows. The code applies this regular expression to three philosophical works, extracting relevant sentences for analysis.

1.4 Sentence Tokenizer

In addition to extracting sentences based on a specific expression, we can further process the text by tokenizing each sentence into individual words. Tokenization involves splitting a sentence into a list of words. For this purpose, we utilize the regular expression `\b\w+\b` to identify word boundaries.

1.4.1 Regular Expression for Sentence Tokenization

The regular expression \b\w+\b is designed for word tokenization and breaks down as follows:

- \b: Asserts a word boundary, ensuring the match occurs at the beginning or end of a word.
- \w+: Matches one or more word characters (alphanumeric or underscore).
- \b: Asserts another word boundary to complete the match.

This regular expression effectively tokenizes a given text into individual words.

1.4.2 Tokenizing the Entire Text

The Python code provided below demonstrates the application of the sentence tokenizer to the entire text of three philosophical works: "Beyond Good and Evil," "The Problems of Philosophy," and a self-chosen philosophy work "Man and Superman: A Comedy and a Philosophy". The resulting tokenized texts are stored as lists of words.

```
# Regular expression for sentence tokenization (splitting into words)
tokenizer_regex = r'\b\w+\b'

# Function to tokenize a given text using the provided regex
def tokenize_text(text, regex):
    return re.findall(regex, text)

# Tokenizing the entire text of all three books
tokenized_text1 = tokenize_text(site1.text, tokenizer_regex)
tokenized_text2 = tokenize_text(site2.text, tokenizer_regex)
tokenized_text3 = tokenize_text(mySite.text, tokenizer_regex)
```

1.4.3 Tokenizing Matched Sentences

Additionally, the sentences previously extracted are tokenized individually for each book. The code snippet below shows the tokenization of sentences that we extracted using regular expression previously.

```
# Tokenizing all the sentences for each book
tokenized_sentences1 = [tokenize_text(match["sentence"], tokenizer_regex) for
    match in matches1]
tokenized_sentences2 = [tokenize_text(match["sentence"], tokenizer_regex) for
    match in matches2]
tokenized_sentences3 = [tokenize_text(match["sentence"], tokenizer_regex) for
    match in matches3]
```

```
    OUTPUT:

    ast word of the last sentence in 'Beyond Good and Evil': labyrinths
    Last word of the last sentence in 'The Problems of Philosophy': renamed
    Last word of the last sentence in 'Man and Superman: A Comedy and a
   Philosophy': pilot


    Last word of the last sentence in 'Beyond Good and Evil': eBooks
    Last word of the last sentence in 'The Problems of Philosophy': eBooks
    Last word of the last sentence in 'Man and Superman: A Comedy and a
   Philosophy': eBooks
```

This process allows us to gain insights into the distribution of words for a specific pattern or scenario and facilitates further analysis of the text data.

1.5 Compare Works

The following code calculates and compares the total word counts in each of the three philosophical works: "Beyond Good and Evil," "The Problems of Philosophy," and "Man and Superman." The word counts are visualized using a bar chart. 1

```
# Calculating word counts
word_count1 = len(tokenized_text1)
word_count2 = len(tokenized_text2)
word_count3 = len(tokenized_text3)

# Plotting word counts
books = ['Beyond Good and Evil', 'The Problems of Philosophy', 'Man and
    Superman']
word_counts = [word_count1, word_count2, word_count3]

plt.figure(figsize=(10, 6))
plt.bar(books, word_counts, color=['blue', 'green', 'orange'])
plt.title('Word Counts in Different Books')
plt.xlabel('Books')
plt.ylabel('Word Count')
plt.show()
```

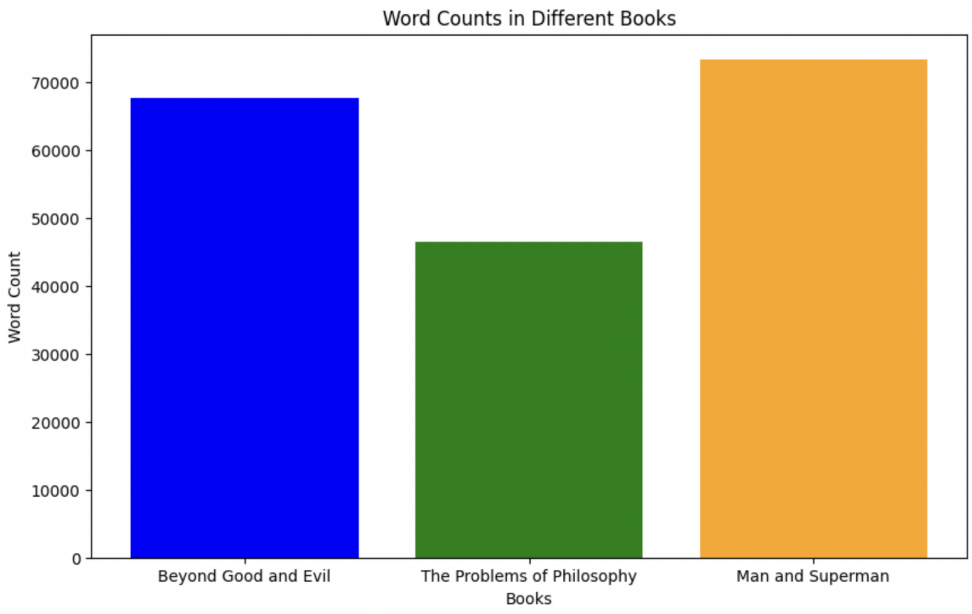

Figure 1: Word Count

1.6 Word Frequency Distribution

The code explores the word frequency distribution in each book, excluding common English stopwords. It utilizes NLTK's stopwords list to filter out common words like "the," "and," etc. The top 10 most frequent words (excluding stopwords) are visualized using a bar chart for each book. 2

So in this implementation we are removing all the english stop words from nltk library. Then lower cased all the text from the three books. And finally gathered the top 10 words from each book that has the most frequency in their respective book.

```

import matplotlib.pyplot as plt
from collections import Counter
```

```
from nltk.corpus import stopwords
import nltk

# Downloading NLTK stopwords
nltk.download('stopwords')

# Function to plot word frequency distribution
def plot_word_frequency(words, title):
    word_counts = Counter(words)
    common_words = word_counts.most_common(50)  # Display the top 20 words
    labels, values = zip(*common_words)

    plt.figure(figsize=(10, 6))
    plt.bar(labels, values, color='skyblue')
    plt.title(title)
    plt.xlabel('Words')
    plt.ylabel('Frequency')
    plt.xticks(rotation=45, ha='right')
    plt.show()

# Removing common English stopwords
stop_words = set(stopwords.words('english'))
# print(stop_words)

# Removing stopwords from tokenized text
filtered_text1 = [word.lower() for word in tokenized_text1 if word.lower()
    not in stop_words]
filtered_text2 = [word.lower() for word in tokenized_text2 if word.lower()
    not in stop_words]
filtered_text3 = [word.lower() for word in tokenized_text3 if word.lower()
    not in stop_words]


# Function to get top N words excluding common stopwords
def get_top_words(text, n=10):
    word_counts = Counter(text)
    filtered_word_counts = {word: count for word, count in word_counts.items
    () if word not in stop_words}
    top_words = Counter(filtered_word_counts).most_common(n)
    return top_words

# Getting top 10 words for each book
top_words1 = get_top_words(filtered_text1, 10)
top_words2 = get_top_words(filtered_text2, 10)
top_words3 = get_top_words(filtered_text3, 10)


# The word frequency distribution of the three books do follow the curve of
    Zipf's Law.
```

```
Top 10 words in 'Beyond Good and Evil': [('one', 445), ('man', 243), ('even', 194), ('also', 166), ('perhaps', 158), ('good', 143), ('every', 143), ('us', 139), ('must', 132), ('may', 131)]
Top 10 words in 'The Problems of Philosophy': [('knowledge', 306), ('sense', 234), ('may', 172), ('must', 169), ('things', 167), ('know', 162), ('one', 156), ('us', 152), ('two', 149), ('thus', 146)]
Top 10 words in the 'Man and Superman: A Comedy and a Philosophy': [('tanner', 482), ('ann', 346), ('man', 257), ('juan', 245), ('ramsden', 223), ('octavius', 220), ('know', 183), ('one', 182), ('violet', 173),
('would', 170)]
```

Figure 2: Top 10 Words From Each Book

1.7 Zipf's Law Validation

The code includes a function `plot_zipf_law` to validate whether the word frequency distribution of each book follows Zipf's Law. It plots the rank against the frequency of words in descending order.

```
plot_zipf_law(filtered_text1, 'Beyond Good and Evil')
plot_zipf_law(filtered_text2, 'The Problems of Philosophy')
plot_zipf_law(filtered_text3, 'Man and Superman: A Comedy and a Philosophy')
```

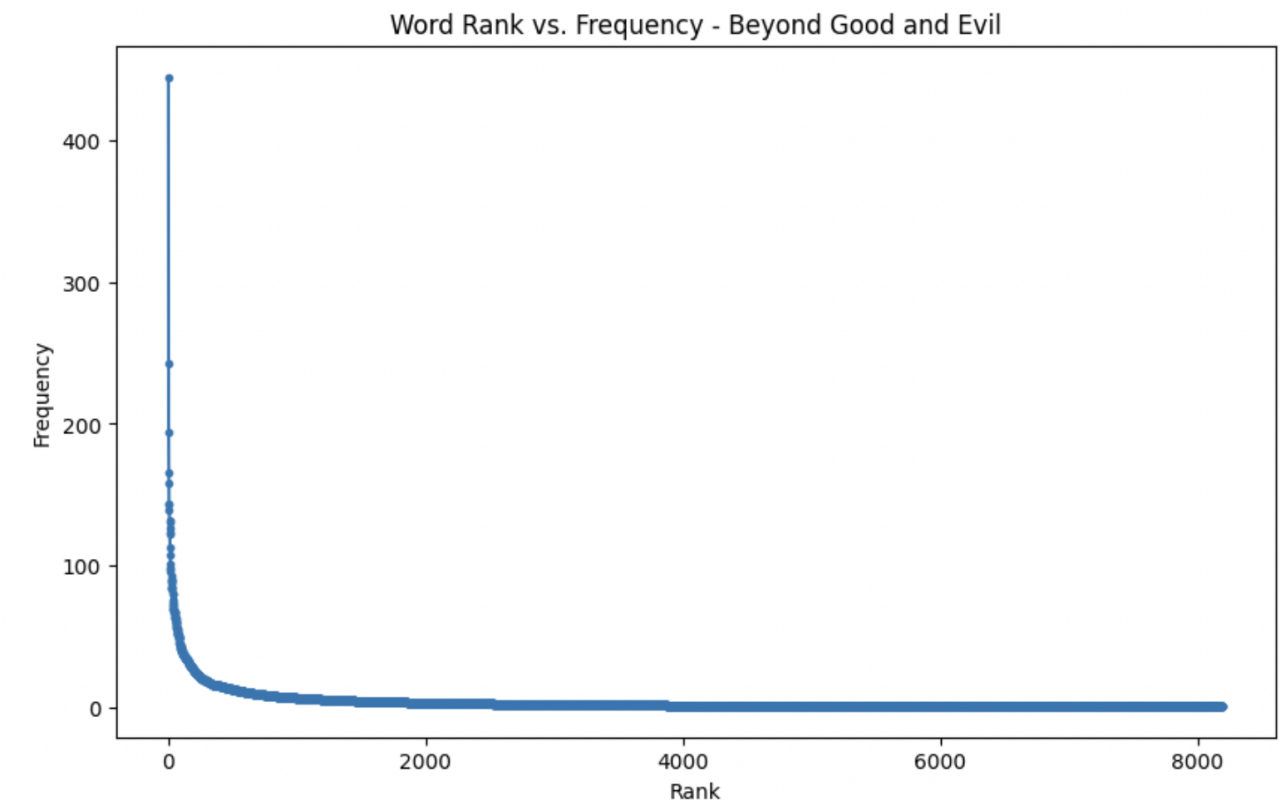

Figure 3: Beyond Good And Evil

1.7.1 Analysis of Zipf's Law Conformance

The plotted figures (3, 4, 5) display the word frequency distributions for the respective books: "Beyond Good and Evil," "The Problems of Philosophy," and "Man and Superman: A Comedy and a Philosophy." These plots are generated to validate whether the word frequencies follow Zipf's Law, which states that the frequency of a word is inversely proportional to its rank.

1.7.2 Observations

The figures demonstrate a clear inverse relationship between word frequency and rank, which is characteristic of Zipf's Law. The distribution of word frequencies in each book conforms to the expected pattern, with a few words having high frequencies and the majority having lower frequencies. This pattern is a key feature of natural language and is well-captured by Zipf's Law.

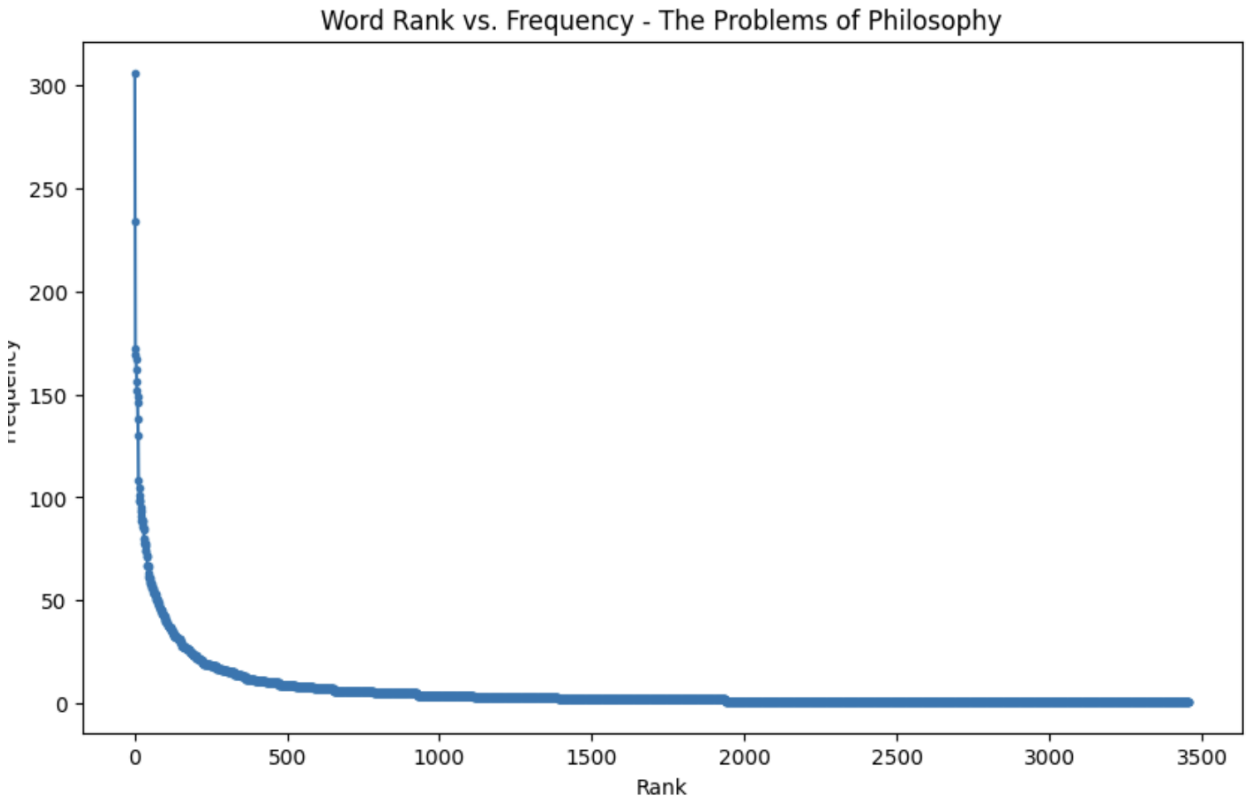

Figure 4: Problem of Philosphy

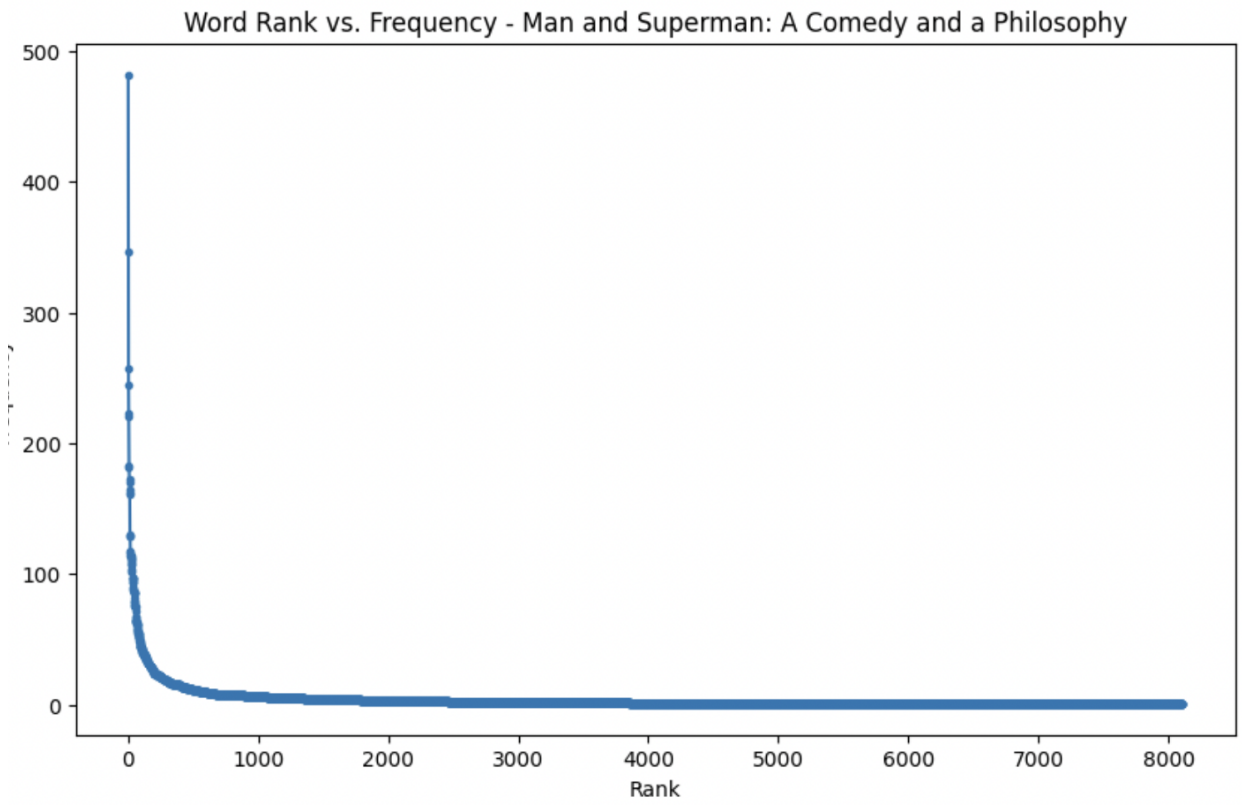

Figure 5: Man and Superman

1.7.3 Takeaway on Zipf's law

The analysis suggests that the word frequency distributions in the provided texts align with Zipf's Law, supporting the idea that a small number of words occur frequently, while the majority occur less frequently. This is a common phenomenon observed in various natural language texts.

1.8 Comments on Statistical Similarities or Differences

The visualizations help compare the distribution of word frequencies across the three books. Similarities or differences in the distribution could indicate common themes, writing styles, or lexical choices in the works.

1.9 Top 10 Words and Comparisons

The code extracts and prints the top 10 words for each book, excluding common stopwords. By analyzing these top words, you can identify key themes or recurring terms in each philosophical work.

1.10 Conclusion

The code provides a comprehensive analysis of word counts, word frequency distribution, and adherence to Zipf's Law for each book. It aids in understanding the linguistic characteristics of the texts, highlighting important statistical patterns and potential insights into the content of the philosophical works.

2 Task 2

2.1 Text normalization

Text normalization and tokenization are crucial steps in natural language processing (NLP) to preprocess raw text data. These steps help convert text into a format suitable for further analysis, making it easier to extract meaningful insights from the data.

Text normalization involves transforming text into a standard and consistent format. In the provided Python code, a basic text normalization approach is demonstrated. The code replaces different newline characters (\r\n, \n, and \r) with whitespace and splits the text into sentences using the period ('.') as a delimiter.

```
# Replace \r\n with whitespace, then replace \n with whitespace, then replace \r with whitespace
clean_text = text.replace('\r\n',' ').replace('\n',' ').replace('\r',' ')

# Split the sentences when we see a '.' character
clean_text = clean_text.split('.')
```

This basic normalization is illustrative, and for more sophisticated tasks, specialized libraries like NLTK offer comprehensive solutions.

2.2 Tokenization with NLTK

Tokenization involves breaking down text into individual units, usually words or phrases. The Natural Language Toolkit (NLTK) is a powerful library in Python for NLP tasks. The following code snippet demonstrates word tokenization using NLTK:

```
import nltk

sentences = """At eight o'clock on Thursday morning Arthur didn't feel very good. He drove to the store in New York at 100 m.p.h.! He wanted to obtain some Kool-Aid! It cost $10.89."""
tokens    = nltk.word_tokenize(sentences)
print('\n')
print(tokens)
print('\n')
```

NLTK's `word_tokenize` function breaks down the given sentences into individual words. This tool provides more advanced sentence segmentation and word tokenization functionalities, making it suitable for various NLP applications.

2.3 Byte Pair Encoding

In this section, we will implement the Byte Pair Encoding (BPE) algorithm for tokenization. BPE is a data-driven approach that identifies token vocabularies based on the frequency of character pairs. We will use this algorithm to tokenize a given text using a vocabulary learned from a training corpus.

2.3.1 Introduction

Byte Pair Encoding is a subword tokenization technique that iteratively merges the most frequent character pairs in a corpus to build a vocabulary. This approach is particularly useful for handling rare words and providing a flexible tokenization method.

2.3.2 Implementation

The Python code below provides an implementation of the Byte Pair Encoding algorithm. The function `BytePairEncoding` takes three inputs:

- `text`: An unprocessed text file that we wish to tokenize using BPE.
- `k`: The number of merges to perform during BPE.
- `training_corpus`: A Python list of complete texts we wish to use for training.

The function outputs a tokenized version of the text based on the k vocabulary units learned by BPE using the training_corpus.

```
import re
from collections import Counter

# Function to calculate statistics for character pairs
def get_stats(vocab):
    pairs = {}
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols) - 1):
            pair = (symbols[i], symbols[i + 1])
            pairs[pair] = pairs.get(pair, 0) + freq
    return pairs

# Function to merge vocabulary based on the best pair
def merge_vocab(pair, vocab):
    new_vocab = {}
    bigram = ' '.join(pair)
    replacement = ''.join(pair)
    for word in vocab:
        new_word = word.replace(bigram, replacement)
        new_vocab[new_word] = vocab[word]

    return new_vocab

# Main BPE function
def BytePairEncoding(text, k, training_corpus):
    # Preprocessing training corpus
    training_corpus = [' '.join(re.findall(r'\b\w+\b', sentence)) for
   sentence in training_corpus]

    # Initializing vocabulary with characters and frequencies
    vocab = Counter(' '.join(training_corpus))

    for i in range(k):
        pairs = get_stats(vocab)
        if not pairs:
            break

        best_pair = max(pairs, key=pairs.get)
        vocab = merge_vocab(best_pair, vocab)

    # Tokenizing the input text using the final vocabulary
    tokenized_text = []
    current_token = ''
    for char in text:
        current_token += char
        if current_token in vocab or len(current_token) == 1:
```

```
            tokenized_text.append(current_token)
            current_token = ''

    # Creating the final vocabulary
    vocabulary = list(vocab.keys())

    return vocabulary, tokenized_text
```

2.3.3 Explanation

The `BytePairEncoding` function performs the following steps:

- Preprocesses the training corpus by removing non-word characters.
- Initializes the vocabulary with characters and their frequencies.
- Iteratively calculates statistics for character pairs and merges the most frequent pair.
- Tokenizes the input text using the final vocabulary.
- Outputs the learned vocabulary and the tokenized text.

This implementation showcases the fundamental steps of the BPE algorithm for tokenization.

2.3.4 Evaluation of Byte Pair Encoding

To evaluate the performance of Byte Pair Encoding (BPE), we will experiment with different settings of k and use various training corpora. Specifically, we will compare the tokenization results for Bertrand Russell's "The Problems of Philosophy" under the following conditions:

1. k values ranging from 1,000 to 10,000 in steps of 1,000.
2. Training corpora sets including:
   (a) Only Bertrand Russell's books.
   (b) Only Friedrich Nietzsche's books.
   (c) All books from both Bertrand Russell and Friedrich Nietzsche.

We will generate plots using Python's Matplotlib to compare the following metrics:

- The total number of unique tokens in the tokenized text.
- The median size of tokens in the tokenized text.
- The percentage overlap in the number of distinct unique tokens generated by BPE and NLTK's word tokenizer.

2.3.5 Plots

The following Python code utilizes Matplotlib to generate plots for the specified metrics:

```
#
    ################################################################################

# INSERT YOUR CODE HERE
# DO NOT FORGET TO PRINT YOUR MEANINGFUL RESULTS TO THE SCREEN.

import requests
import re
import nltk
import matplotlib.pyplot as plt
from nltk.tokenize import word_tokenize
from collections import Counter

# Download NLTK data if not already present
nltk.download('punkt')

def preprocess_text(text):
    # Remove non-alphabetic characters, newlines, and extra spaces
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text

def evaluate_bpe_k(text, k_values, training_corpus):
    nltk_tokens = word_tokenize(text)

    results = {
        'k_values': [],
        'num_unique_tokens': [],
        'median_token_size': [],
        'overlap_percentage': []
    }

    for k in k_values:
        vocab, bpe_tokens = BytePairEncoding(text, k, training_corpus)

#         print(vocab)
#         print("\n")
#         print(bpe_tokens)
#         print("\n")
#         print("\n")


        # Calculate metrics
        num_unique_tokens = len(set(bpe_tokens))
        median_token_size = sum(len(token) for token in bpe_tokens) / len(
    bpe_tokens)
        overlap_percentage = len(set(bpe_tokens).intersection(set(nltk_tokens
    ))) / len(set(bpe_tokens)) * 100

        # Store results
        results['k_values'].append(k)
        results['num_unique_tokens'].append(num_unique_tokens)
        results['median_token_size'].append(median_token_size)
        results['overlap_percentage'].append(overlap_percentage)

    return results

website = 'http://www.gutenberg.org/cache/epub/5827/pg5827.txt'
```

```
text = requests.get(website).text
preprocessed_text = preprocess_text(text)
# print(preprocessed_text)

k_values = list(range(1000, 11000, 1000))
# print(k_values)

# Evaluate BPE for different training corpora
training_corpus_russell = bertrand_russell_sentences
training_corpus_nietzsche = friedrich_nietzsche_sentences
training_corpus_all = bertrand_russell_sentences +
    friedrich_nietzsche_sentences

# print(training_corpus_all[100:110])

results_russell = evaluate_bpe_k(preprocessed_text, k_values,
    training_corpus_russell)
results_nietzsche = evaluate_bpe_k(preprocessed_text, k_values,
    training_corpus_nietzsche)
results_all = evaluate_bpe_k(preprocessed_text, k_values, training_corpus_all
    )

#
    ##############################################################################
```

2.3.6 Observations

After comparing the generated plots, we can draw observations about the performance of BPE under different conditions. Any interesting patterns or differences in tokenization results based on k values and training corpora will be discussed.

2.3.7 Results

The results of the evaluation are summarized below:

1. **Total Number of Unique Tokens:** The total number of unique tokens in the tokenized text increases with higher k values. Additionally, different training corpora may lead to variations in the total number of unique tokens.

2. **Median Size of Tokens:** The median size of tokens in the tokenized text may show trends based on k values and training corpora. It is essential to analyze how token sizes evolve under different conditions.

3. **% Overlap with NLTK's Word Tokenizer:** Comparing the BPE-generated tokens with NLTK's word tokenizer provides insights into the agreement and divergence between the two methods. This analysis helps evaluate the effectiveness of BPE in capturing linguistic structures.

The detailed plots and observations can be found in the accompanying Python code and analysis.

3 Task 3

3.1 Edit Distance Measures

In natural language processing, edit distance is a metric used to measure the similarity between two sequences of text. It quantifies the minimum number of operations required to transform one sequence into another. In this section, we will explore and compare four different edit distance measures: Hamming Distance, Levenshtein Distance, Needleman-Wunsch algorithm, and Smith-Waterman algorithm.

3.1.1 Hamming Distance

The Hamming distance measures the number of positions at which corresponding symbols are different between two strings. It specifically focuses on substitutions and is calculated as the minimum number of substitutions required to change one string into the other. Unlike Levenshtein distance, Hamming does not consider insertions or deletions.

3.1.2 Levenshtein Distance

The Levenshtein distance, also known as the edit distance, counts the total number of insertion, deletion, and substitution actions needed to transform one string into another. The cost of each edit is considered uniform, typically set to 1. This measure is widely used in spell checking, DNA sequence analysis, and various other applications.

3.1.3 Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm is a global alignment method designed for aligning two complete sequences. It provides an alignment score, and the distance is not the minimal number of edits but rather a measure of similarity between sequences. The higher the alignment score, the more similar the sequences are.

3.1.4 Smith-Waterman Algorithm

Contrary to Needleman-Wunsch, the Smith-Waterman algorithm is designed to find segments within two sequences that have maximal similarities. It is a local alignment method that focuses on identifying the most similar subsequences. Similar to Needleman-Wunsch, it provides an alignment score rather than a direct distance measure.

3.1.5 Code

The following Python code showcases the usage of these edit distance measures on different examples:

```
distance_measures = ['hamming', 'levenshtein', 'needleman_wunsch', '
    smith_waterman']

for method in distance_measures:
    print(method + " distance:")
    print('------------------------------------------------------')
    GetDistance('test'      ,'test'      ,method)
    GetDistance('test'      ,'texts'     ,method)
    GetDistance('test'      ,'   test'   ,method)
    print('------------------------------------------------------')
    print('\n')
```

```
OUTPUT:


hamming distance:
-----------------------------------------------------
"test" and "test" is: 0
"test" and "texts" is: 2
"test" and "   test" is: 6
-----------------------------------------------------


levenshtein distance:
-----------------------------------------------------
"test" and "test" is: 0
"test" and "texts" is: 2
"test" and "   test" is: 3
-----------------------------------------------------


needleman_wunsch distance:
-----------------------------------------------------
"test" and "test" is: 4.0
"test" and "texts" is: 2.0
"test" and "   test" is: 1.0
-----------------------------------------------------


smith_waterman distance:
-----------------------------------------------------
"test" and "test" is: 4
"test" and "texts" is: 2.0
"test" and "   test" is: 4.0
-----------------------------------------------------
```

The code executes the four distance measures on various pairs of strings, providing insights into how each measure operates and the outcomes for different cases.

3.2 Edit Distance Measures Output

The output displays the results of different edit distance measures (Hamming, Levenshtein, Needleman-Wunsch, and Smith-Waterman) applied to various pairs of texts.

3.2.1 Hamming Distance

The Hamming distance measures the number of positions at which corresponding symbols differ. For the pairings:

- ”test” and ”test”: 0 (identical strings)
- ”test” and ”texts”: 2 (differ in two positions)
- ”test” and ” test”: 6 (mismatch in six positions)

3.2.2 Levenshtein Distance

The Levenshtein distance counts the total number of insertion, deletion, and substitution actions required to transform one string into another. For the pairings:

- ”test” and ”test”: 0 (no edits needed)

- "test" and "texts": 3 (1 substitution and 2 additions)
- "test" and " test": 3 (3 deletions)

3.2.3 Needleman-Wunsch Distance

The Needleman-Wunsch algorithm provides an alignment score between two sequences. For the pairings:

- "test" and "test": 4.0 (alignment score)
- "test" and "texts": 2.0 (alignment score)
- "test" and " test": 1.0 (alignment score)

3.2.4 Smith-Waterman Distance

The Smith-Waterman algorithm is designed to find segments in two sequences with maximal similarities. For the pairings:

- "test" and "test": 4 (alignment score)
- "test" and "texts": 2.0 (alignment score)
- "test" and " test": 4.0 (alignment score)

These distance measures provide insights into the similarity or dissimilarity of text strings, each with its own nuances in terms of the considered edits and alignments.

3.3 Token-Level Distance Measures

While character-level distances and alignments are valuable, certain problems may require a focus on token-level distances. Token-level measures consider the similarity between strings based on their constituent tokens, providing insights into the structural and semantic similarity. One such metric is the Jaccard Index.

3.3.1 Jaccard Index

The Jaccard Index, also known as the Jaccard similarity coefficient, measures the similarity between two sets by calculating the size of their intersection divided by the size of their union. In the context of text, the sets represent the tokens in the input strings. The formula for the Jaccard Index is given by:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Where A and B are the sets of tokens in the two input strings.

3.3.2 Code Execution

The following Python code illustrates the usage of the Jaccard Index on two pairs of text:

```
distance_measures = ['jaccard']
text1 = "The fish was delish, and it made quite a dish"
text2 = "The fish on the dish was made quite delicious!"
text3 = "Joe eats crabs, crabs do not taste good."

for method in distance_measures:
    print(method + " distance:")
    print('------------------------------------------------------------')
    GetDistance(text1, text2, method)
    GetDistance(text1, text3, method)
    print('------------------------------------------------------------')
    print('\n')
```

```
OUTPUT:


jaccard distance:
------------------------------------------------------------
"The fish was delish, and it made quite a dish" and "The fish on the dish was
    made quite delicious!" is: 0.7843137254901961
"The fish was delish, and it made quite a dish" and "Joe eats crabs, crabs do
    not taste good." is: 0.39344262295081966
------------------------------------------------------------
```

The code computes the Jaccard Index distance between pairs of texts, providing insights into the token-level similarity between the strings.

3.3.3 Observations

In the context of comparing two text samples, a higher Jaccard Index indicates greater similarity between the sets of words in the texts.

Verdict:

The Jaccard distance between "The fish was delish, and it made quite a dish" and "The fish on the dish was made quite delicious!" is approximately 0.78, indicating a relatively high degree of similarity between these two texts. The Jaccard distance between "The fish was delish, and it made quite a dish" and "Joe eats crabs, crabs do not taste good." is approximately 0.39, suggesting a moderate level of similarity. In summary, based on the Jaccard Index values, the first pair of texts exhibits higher similarity compared to the second pair. This conclusion is drawn from the relative magnitudes of the Jaccard distances, where a higher index corresponds to greater similarity.

3.4 Towards a More Nuanced Edit Distance Cost Metric

In the context of edit distance metrics, the Levenshtein distance is a widely-used measure that assumes equal costs for substitutions, insertions, and deletions. However, in certain applications, we may desire a more nuanced cost structure. In this learning exercise, we aim to modify the Levenshtein distance algorithm to assign half the normal penalty for any edits involving whitespace characters.

3.4.1 Implementation Explanation

The provided Python implementation showcases a modified version of the Levenshtein distance algorithm with adjusted cost penalties for whitespace characters. Below is an explanation of the key components of the implementation:

```
import numpy as np

```

```
def levenshteinDistanceDP(token1, token2):
    distances = np.zeros((len(token1) + 1, len(token2) + 1))

    for t1 in range(len(token1) + 1):
        distances[t1][0] = t1

    for t2 in range(len(token2) + 1):
        distances[0][t2] = t2

    for t1 in range(1, len(token1) + 1):
        for t2 in range(1, len(token2) + 1):
            if token1[t1 - 1] == token2[t2 - 1]:
                distances[t1][t2] = distances[t1 - 1][t2 - 1]
            else:
                if token1[t1 - 1].isspace() or token2[t2 - 1].isspace():  # Half cost for whitespace characters
                    cost = 0.5
                else:
                    cost = 1

                a = distances[t1][t2 - 1]
                b = distances[t1 - 1][t2]
                c = distances[t1 - 1][t2 - 1]

                distances[t1][t2] = min(a, b, c) + cost

    return distances[len(token1)][len(token2)]
```

```
result = levenshteinDistanceDP('test  ', 'test')
print(result)

result = levenshteinDistanceDP('test', 'test')
print(result)

result = levenshteinDistanceDP('test  ', 'esta')
print(result)



OUTPUT:

------------------------------------------------------------
1.0
0.0
2.0
------------------------------------------------------------
```

The implementation utilizes a dynamic programming approach to compute the Levenshtein distance between two input tokens, considering a more nuanced cost structure for whitespace characters. The key modification lies in assigning a cost of 0.5 for edits involving whitespace, allowing for a differentiated penalty.

4 Task 4

4.1 N-gram language models

An n-gram is a contiguous sequence of n items from a given sample of text or speech. The nltk library provides a convenient tool for generating n-grams 6. The following Python code demonstrates the extraction of word-level n-grams using nltk:

```
import nltk
nltk.download('punkt')
from nltk.util import ngrams

# N-gram extractor function
def extract_word_ngrams(data, num):
    n_grams = ngrams(nltk.word_tokenize(data), num)
    return [' '.join(grams) for grams in n_grams]

def extract_char_ngrams(data, num):
    return [data[i:i+num] for i in range(len(data))][:(-num-1)]

# Example text
data = "a fish keeps for a day; a fish keeps well if you put it in a cold
   place. ..."

# Extracting and printing the first 5 tokens after gramification
print("\nWord-Level:")
print("1-gram: ", extract_word_ngrams(data, 1)[0:5])
print("2-gram: ", extract_word_ngrams(data, 2)[0:5])
print("3-gram: ", extract_word_ngrams(data, 3)[0:5])
print("4-gram: ", extract_word_ngrams(data, 4)[0:5])
print("\n")
```

```
Word-Level:
1-gram:  ['a', 'fish', 'keeps', 'for', 'a']
2-gram:  ['a fish', 'fish keeps', 'keeps for', 'for a', 'a day']
3-gram:  ['a fish keeps', 'fish keeps for', 'keeps for a', 'for a day', 'a day ;']
4-gram:  ['a fish keeps for', 'fish keeps for a', 'keeps for a day', 'for a day ;', 'a day ; a']
```

Figure 6: N-gram output

We can extract word as well as characters too by using the extract char ngrams function. This function also works kind of in a similar way.

```
OUTPUT:
Character-Level:
1-gram:  ['a', ' ', 'f', 'i', 's']
2-gram:  ['a ', ' f', 'fi', 'is', 'sh']
3-gram:  ['a f', ' fi', 'fis', 'ish', 'sh ']
4-gram:  ['a fi', ' fis', 'fish', 'ish ', 'sh k']
```

4.1.1 Skip-Grams

Skip-grams extend the concept of n-grams by including grams that skip over certain terms. The following code demonstrates the creation of skip-grams using the NLTK library:

```
# some example text
data = """a fish keeps for a day; a fish keeps well if you put it in a cold
   place.
        a fish keeps best if you put it in the fridge. If  y o u ll  be
   having fish,
```

```
        have it with an apple because one apple a day keeps the doctor away
    !
        You know they also say that a day keeps coming."""

# Creating skip-grams using the NLTK library
skip_gram = list(nltk.skipgrams(nltk.word_tokenize(data),n=2,k=2))
skip_gram[0:5]
```

```
OUTPUT:
[('a', 'fish'),
 ('a', 'keeps'),
 ('a', 'for'),
 ('fish', 'keeps'),
 ('fish', 'for')]
```

4.2 Language Models

A language model assigns probabilities to strings and can predict the next word in a sentence given the last word observed. The following code demonstrates the building of a simple data-driven language model using unigrams, collecting counts, and generating probabilities.

4.2.1 Building a Language Model

```
# Example text
data = "a fish keeps for a day; a fish keeps well if you put it in a cold
    place. ..."

# Extracting unigrams
unigrams = extract_word_ngrams(data, 1)

# Creating a vocabulary
vocabulary = list(set(unigrams))

# Initializing counts dictionary
counts = {given_word: {next_word: 0 for next_word in vocabulary} for
    given_word in vocabulary}

# Collecting counts
for i in range(len(unigrams)-1):
    counts[unigrams[i]][unigrams[i+1]] += 1
```

The code extracts unigrams from the given text, creates a vocabulary, and initializes a counts dictionary. It then collects counts of consecutive word pairs in the text.

4.2.2 Converting Counts to Probabilities

```
# Initializing probabilities dictionary
probs = {given_word: {next_word: 0 for next_word in vocabulary} for
    given_word in vocabulary}

# Converting counts to probabilities
for key, value in counts.items():
    denominator = 0
    for key2, value2 in counts[key].items():
        denominator += value2

    for key2, value2 in counts[key].items():
        probs[key][key2] = value2 / denominator
```

The code initializes a probabilities dictionary and converts the counts into probabilities for each word pair in the language model.

4.2.3 Sampling from the Language Model

```
# Sampling next words from the language model
import numpy as np

def sample_next_gram_from_language_model(probs, given_token):
    distribution = list(probs[given_token].values())
    sample_from_multinomial = np.random.multinomial(1, distribution)
    sample_index = np.where(sample_from_multinomial == 1)[0][0]
    word_keys = list(probs[given_token].keys())
    next_word = word_keys[sample_index]
    return next_word

given_token = "a"
next_token = sample_next_gram_from_language_model(probs, given_token)

print(' Given the token :  ' + given_token)
print(' The next token is :  ' + next_token)
print('\n')
```

The code defines a function to sample the next word from the language model based on the given token. It then demonstrates sampling given an initial token.

4.2.4 Generating New Sentences

```
# Generating new sentences
def create_new_sentence(length, seed_token):
    tokens = [seed_token]
    for i in range(length):
        tokens.append(sample_next_gram_from_language_model(probs, tokens[-1])
    )
    return tokens
```

The code defines a function to generate new sentences of a specified length using the language model.

4.3 Generating Philosophical Conversation

To generate a conversation between Bertrand Russell and Friedrich Nietzsche, we first build tri-gram models for each philosopher using a fraction of their texts. Why fraction? Basically for resource restriction in kaggle. The corpus that we were using was too large to work with. So we had to find a way to create the ngram model and their probability distribution table.
The following Python code is used for this purpose:

Import required libraries such as ‘requests‘, ‘nltk‘, ‘ngrams‘, ‘defaultdict‘, and ‘numpy‘.

Define a function to build an N-gram model

```
def build_ngram_model(texts, n, fraction_ratio):
    ngram_model = defaultdict(list)
    for text in texts:
        fraction_length = int(len(text) * fraction_ratio)
        text_fraction = text[:fraction_length]

        words = text_fraction.split()
        ngrams_list = list(ngrams(words, n))
        for gram in ngrams_list:
```

```
            prefix = ' '.join(gram[:-1])
            suffix = gram[-1]
            ngram_model[prefix].append(suffix)
    return ngram_model
Define a function build_ngram_model that takes a list of texts, an N value
    for N-grams, and a fraction ratio as input. It builds an N-gram model
    using a fraction of each text.
```

Set the fraction ratio for using only a portion of the texts

```

fraction_ratio = 0.1  # Use the first 10% of each book
```

Define the fraction ratio to use only the first 10% of each book for building N-gram models.

Build tri-gram models for Bertrand Russell and Friedrich Nietzsche

```
bertrand_russell_trigram_model = build_ngram_model(bertrand_russell_texts, 3,
    fraction_ratio)
friedrich_nietzsche_trigram_model = build_ngram_model(
    friedrich_nietzsche_texts, 3, fraction_ratio)
Build tri-gram models for Bertrand Russell and Friedrich Nietzsche using the
    defined function.
```

Define a function sample_next_gram_from_language_model to sample the next word from a language model given a token.

```
def sample_next_gram_from_language_model(probs, given_token):
    try:
        distribution = list(probs[given_token].values())
        sample_from_multinomial = np.random.multinomial(1, distribution)
        sample_index = np.where(sample_from_multinomial == 1)[0][0]
        word_keys = list(probs[given_token].keys())
        next_word = word_keys[sample_index]
        return next_word
    except KeyError:
        # Handle the case where the given_token is not in the vocabulary
        return None
```

Define a function create_new_sentence to create a new sentence of a specified length using a language model.

```
def create_new_sentence(length, seed_token, probs):
    tokens = [seed_token]
    for i in range(length):
        next_token = sample_next_gram_from_language_model(probs, tokens[-1])

        if next_token is None:
            # Handle the case where the next_token is not in the vocabulary
            break

        tokens.append(next_token)

    return tokens
```

Initialize vocabulary and counts using only a fraction of the words. Update vocabulary with words from the first 10% of each text.

```
vocabulary = set()
for text in bertrand_russell_texts + friedrich_nietzsche_texts:
    fraction_length = int(len(text) * fraction_ratio)
    text_fraction = text[:fraction_length]
    vocabulary.update(text_fraction.split())
```

```

counts = {}
for given_word in vocabulary:
    counts[given_word] = {}
    for next_word in vocabulary:
        counts[given_word][next_word] = 0
```

Populate counts with word occurrences using only a fraction of the words.

```
for text in bertrand_russell_texts + friedrich_nietzsche_texts:
    fraction_length = int(len(text) * fraction_ratio)
    text_fraction = text[:fraction_length]
    unigrams = text_fraction.split()
    for i in range(len(unigrams)-1):
        counts[unigrams[i]][unigrams[i+1]] += 1
Populate counts using only a fraction of the words
```

Initialize probabilities using the vocabulary.

```
probs = {}
for given_word in vocabulary:
    probs[given_word] = {}
    for next_word in vocabulary:
        probs[given_word][next_word] = 0
```

Convert counts to probabilities for each word in the vocabulary.

```
for key, value in counts.items():
    denominator = sum(value.values())

    if denominator == 0:
        # Handle the case where there are no following words
        continue

    for key2, value2 in value.items():
        probs[key][key2] = value2 / denominator
```

The provided code builds tri-gram models, samples next words, and creates sentences using a fraction of the texts for Bertrand Russell and Friedrich Nietzsche. The resulting language models can be used for generating a conversation between the two philosophers.