Max Liu

Professor Hernandez

CISB 63

3 December 2023

Final Text-Summarization Project

Explanation of the Project:

The Automated Text Summarization Tool is designed to provide a concise summary of lengthy textual documents using natural language processing (NLP) techniques. The project incorporates various NLP methodologies to analyze and extract key information from the input text, enabling users to quickly grasp the main ideas without reading the entire document.

NLP Libraries and PreTrained Models: Chose 2

1. **Utilizing SpaCy**

```
# NLP using SpaCy (Named Entity Recognition)
nlp = spacy.load("en_core_web_sm")
doc = nlp(pdf_text)

# Extracting entities and their labels
entities = [(ent.text, ent.label_) for ent in doc.ents]

# Counting the occurrences of each entity label
label_counts = {}
for _, label in entities:
    label_counts[label] = label_counts.get(label, 0) + 1
```

```
# Sorting the labels based on their counts
sorted_labels = sorted(label_counts.items(), key=lambda x: x[1], reverse=True)
```

I utilized SpaCy, a natural language processing library, to perform Named Entity Recognition (NER) on a given text extracted from a PDF document. The code first loads the SpaCy English model and processes the text through SpaCy's NLP pipeline. It then extracts entities and their corresponding labels from the processed document. The entities and labels are stored in a list, and the code proceeds to count the occurrences of each entity label. Finally, it sorts the entity labels based on their frequency in descending order, providing a clear overview of the most frequently occurring named entities in the document. I then used the list and graphed the occurrences of each entity.

2. **Textblob Sentiment Analysis**

```
# Sentiment analysis using TextBlob
blob = TextBlob(pdf_text)

# Get sentiment polarity (-1 to 1, where -1 is negative, 0 is neutral, and 1 is positive)
polarity = blob.sentiment.polarity

# Get sentiment subjectivity (0 to 1, where 0 is very objective and 1 is very subjective)
subjectivity = blob.sentiment.subjectivity

# Print the sentiment analysis results
print("Sentiment Polarity:", polarity)
print("Sentiment Subjectivity:", subjectivity)
```

The code employs TextBlob, a Python library for processing textual data, to conduct sentiment analysis on a given text extracted from a PDF document. The TextBlob class is

used to create a TextBlob object, which represents the text along with its linguistic properties. The sentiment analysis is conducted through the sentiment attribute of the TextBlob object. The polarity attribute provides a numerical value between -1 and 1, where -1 indicates a negative sentiment, 0 signifies a neutral sentiment, and 1 represents a positive sentiment. Similarly, the subjectivity attribute returns a value between 0 and 1, with 0 indicating a highly objective statement and 1 indicating a highly subjective statement. The code then prints out the sentiment polarity and subjectivity scores, providing a quantitative assessment of the sentiment expressed in the text. I then used the word sentiment analysis to graph.

NLP Topics and Methods: Chose 5

3. **Using SpaCy NER to show entities**

```
# NLP using SpaCy (Named Entity Recognition)
nlp = spacy.load("en_core_web_sm")
doc = nlp(pdf_text)

# Named Entity Recognition (NER)
ner_entities = [(ent.text, ent.label_) for ent in doc.ents]

# Display the Named Entities
for entity, label in ner_entities:
    print(f"Entity: {entity}, Label: {label}")
```

The Code leverages the SpaCy library to perform Named Entity Recognition (NER) in the realm of Natural Language Processing (NLP). After loading the SpaCy English

model, the code processes the given text from a PDF document using the model's NLP pipeline, creating a document object (doc). Subsequently, it employs NER to extract named entities along with their respective labels, storing the results in a list named ner_entities. The code then iterates through this list, displaying each named entity along with its associated label. This approach facilitates the identification and categorization of entities such as persons, organizations, and locations within the text, enhancing the understanding of the document's content.

4. **Tokenization**

```
# Tokenization using NLTK
tokens = word_tokenize(pdf_text)
```

The code utilizes the word_tokenize function from the NLTK library to tokenize the pdf_text into a list of individual words, referred to as tokens. Tokenization is the process of breaking down a text into its constituent units, typically words or phrases, to facilitate further analysis or processing. Although it is quite short, I combine it later with more methods to make the project better

5. **POS**

```
# Part-of-Speech (POS) tagging using NLTK with Penn Treebank POS tagset
pos_tags = pos_tag(tokens)


#bruh
pos_mapping = {
    'CC': 'Coordinating conjunction',
    'CD': 'Cardinal number',
    'DT': 'Determiner',
    'EX': 'Existential there',
    'FW': 'Foreign word',
    'IN': 'Preposition or subordinating conjunction',
    'JJ': 'Adjective',
    'JJR': 'Adjective, comparative',
    'JJS': 'Adjective, superlative',
    'LS': 'List item marker',
```

```
    'MD': 'Modal',
    'NN': 'Noun, singular or mass',
    'NNS': 'Noun, plural',
    'NNP': 'Proper noun, singular',
    'NNPS': 'Proper noun, plural',
    'PDT': 'Predeterminer',
    'POS': 'Possessive ending',
    'PRP': 'Personal pronoun',
    'PRP$': 'Possessive pronoun',
    'RB': 'Adverb',
    'RBR': 'Adverb, comparative',
    'RBS': 'Adverb, superlative',
    'RP': 'Particle',
    'SYM': 'Symbol',
    'TO': 'to',
    'UH': 'Interjection',
    'VB': 'Verb, base form',
    'VBD': 'Verb, past tense',
    'VBG': 'Verb, gerund or present participle',
    'VBN': 'Verb, past participle',
    'VBP': 'Verb, non-3rd person singular present',
    'VBZ': 'Verb, 3rd person singular present',
    'WDT': 'Wh-determiner',
    'WP': 'Wh-pronoun',
    'WP$': 'Possessive wh-pronoun',
    'WRB': 'Wh-adverb',
    '$': 'Dollar sign',
    '#': 'Number sign',
    "'": 'Closing quotation mark',
    '(': 'Opening parenthesis',
    ')': 'Closing parenthesis',
    ',': 'Comma',
    '.': 'Period',
    ':': 'Colon',
    '``': 'Opening quotation mark'
}

# Replace POS tags with their corresponding words
pos_tags_readable = [(word, pos_mapping[tag]) for word, tag in
pos_tags]

# Display POS tags along with words
for word, tag in pos_tags_readable[:10]:  # Displaying the first 10
words as an example
    print(f"Word: {word}, POS Tag: {tag}")

# Count the occurrences of each POS tag
pos_counter = Counter(tag for word, tag in pos_tags_readable)
```

I Made the middle a bit smaller so it could be less in the way of the rest of the code, but in this piece of code, I employed the Natural Language Toolkit (NLTK) to perform Part-of-Speech (POS) tagging on a list of tokens extracted from a PDF document. Using the Penn Treebank POS tagset, the code assigns POS tags to each token, and

subsequently, a mapping is established to replace these tags with their corresponding human-readable descriptions. The resulting list provides a more interpretable representation of the text's grammatical elements. The code then displays the first 10 words along with their readable POS tags and counts the occurrences of each POS tag in the document. POS tagging is fundamental in linguistic analysis, offering insights into the syntactic structure of the text and facilitating subsequent natural language processing tasks.I later used the 10 tags and graphed them.

6. **BOW and graph**

```
# Remove stopwords
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
filtered_tokens = [word.lower() for word in tokens if word.isalpha()
and word.lower() not in stop_words]

# Combine the words into a single string
text_for_bow = ' '.join(filtered_tokens)

# Create a Bag-of-Words representation using scikit-learn
vectorizer = CountVectorizer()
bow_matrix = vectorizer.fit_transform([text_for_bow])

# Convert the BOW matrix to a DataFrame for better visualization
bow_df = pd.DataFrame(bow_matrix.toarray(),
columns=vectorizer.get_feature_names_out())

# Display the BOW DataFrame
print("Bag-of-Words (BOW) Representation:")
print(bow_df)
```

The provided Python code focuses on preprocessing text data by removing stopwords, which are common words that typically do not contribute significant meaning to the text. Using the NLTK library, a set of English stopwords is employed, and the original list of tokens is filtered to exclude stopwords and non-alphabetic words. The resulting filtered tokens are then combined into a single string, which serves as input for creating a Bag-of-Words (BOW) representation using scikit-learn's CountVectorizer. This BOW matrix, indicating the frequency of each word in the document, is converted into a DataFrame for improved visualization. The final output is a DataFrame, bow_df, displaying the BOW representation with words as columns and their respective frequencies in the document. This preprocessing step is valuable for transforming raw text into a format suitable for various machine learning and natural language processing tasks.

7. **Word Cloud using results from BOW and Tokenization**

```
# Combine the words into a single string
text_for_wordcloud = ' '.join(filtered_tokens)

# Generate the word cloud
wordcloud = WordCloud(width=800, height=400, random_state=21,
max_font_size=110,
background_color='white').generate(text_for_wordcloud)

# Plot the WordCloud image
plt.figure(figsize=(10, 7))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis('off')
plt.show()
```

This Python code generates a Word Cloud visualization from a filtered set of tokens after removing common stopwords from the text. The filtered tokens are combined into a single string, and a Word Cloud image is created using the WordCloud library, specifying parameters such as width, height, and background color. The resulting Word Cloud visually represents the frequency of words in the text, with larger and bolder words indicating higher occurrence. This intuitive visualization offers a quick overview of the most prominent terms in the document, facilitating the identification of key themes and emphasizing the words that contribute significantly to the content.

And Finally, NLP Application: Summarizing the text. (I only summarized the first 100 sentences since the whole book will take too long to run)

```
# Tokenize the sentences into words
tokenized_sentences = [word_tokenize(sentence.lower()) for sentence in sentences]

# Create a TF-IDF matrix
vectorizer = TfidfVectorizer(max_features=500)  # Limit the number of features
tfidf_matrix = vectorizer.fit_transform([' '.join(sentence) for sentence in
tokenized_sentences])

# Calculate the similarity matrix using dot product
similarity_matrix = tfidf_matrix * tfidf_matrix.T

# Create a graph manually
G = nx.Graph()
num_sentences = len(sentences)
for i in range(num_sentences):
    for j in range(i + 1, num_sentences):
        similarity_score = similarity_matrix[i, j]
        G.add_edge(i, j, weight=similarity_score)

# Calculate TextRank scores for each sentence
```

```
scores = nx.pagerank(G)

# Sort the sentences based on their scores
sorted_sentences = sorted(((scores[i], sentence) for i, sentence in
enumerate(sentences)), reverse=True)

# Extract the top N sentences for the summary
top_n = 3  # Adjust as needed
summary_sentences = [sentence for _, sentence in sorted_sentences[:top_n]]

# Print the summary
print("\n".join(summary_sentences))
```

Explanation Line by Line:

```
# Tokenize the sentences into words
tokenized_sentences = [word_tokenize(sentence.lower()) for sentence in sentences]
```

This line tokenizes each sentence into words using the NLTK word_tokenize function. The lower() method is used to convert all words to lowercase, ensuring consistency in word representation.

```
# Create a TF-IDF matrix
vectorizer = TfidfVectorizer(max_features=500)  # Limit the number of features
tfidf_matrix = vectorizer.fit_transform([' '.join(sentence) for sentence in
tokenized_sentences])
```

Here, a TF-IDF (Term Frequency-Inverse Document Frequency) matrix is created using scikit-learn's TfidfVectorizer. It converts the tokenized sentences into a numerical matrix, where each row corresponds to a sentence, and the columns represent unique words. The max_features parameter limits the number of features (words) included in the matrix to 500.

```
# Calculate the similarity matrix using dot product
similarity_matrix = tfidf_matrix * tfidf_matrix.T
```

The code calculates a similarity matrix by taking the dot product of the TF-IDF matrix and its transpose. This matrix represents the pairwise similarity between sentences based on their TF-IDF representations.

```
# Create a graph manually
G = nx.Graph()
num_sentences = len(sentences)
for i in range(num_sentences):
    for j in range(i + 1, num_sentences):
        similarity_score = similarity_matrix[i, j]
        G.add_edge(i, j, weight=similarity_score)
```

A graph (networkx.Graph) is created where each sentence is a node, and edges between nodes represent the similarity between sentences. The weight of each edge is set to the corresponding similarity score from the similarity matrix.

```
# Calculate TextRank scores for each sentence
scores = nx.pagerank(G)
```

TextRank scores are calculated for each sentence using the PageRank algorithm from the networkx library. These scores represent the importance of each sentence in the document.

```
# Sort the sentences based on their scores
sorted_sentences = sorted(((scores[i], sentence) for i, sentence in
enumerate(sentences)), reverse=True)
```

The sentences are sorted based on their TextRank scores in descending order.

```
# Extract the top N sentences for the summary
top_n = 3
summary_sentences = [sentence for _, sentence in sorted_sentences[:top_n]]
```

The top N sentences with the highest TextRank scores are extracted to form the summary. In this example, the variable top_n is set to 3, meaning the summary consists of the top three sentences.

```
# Print the summary
print("\n".join(summary_sentences))
```

The summary sentences are printed to the console, with each sentence on a new line.

Conclusion:

In conclusion, this NLP project has been an engaging exploration into the realm of natural language processing, offering a thorough analysis of a novel presented in PDF format. Through the utilization of diverse techniques and libraries such as spaCy, TextBlob, NLTK, and scikit-learn, I successfully navigated tasks ranging from text extraction and data cleaning to exploratory data analysis. Notable achievements include named entity recognition, sentiment analysis, part-of-speech tagging, tokenization, and the creation of a Bag-of-Words model. I also delved into advanced techniques like TF-IDF and TextRank for more nuanced analyses, culminating in the generation of a meaningful summary based on sentence similarity. Visualizations with tools like Matplotlib and WordCloud added an extra layer of insight. In essence, this project underscored the versatility and efficacy of NLP, unveiling profound insights from the textual data and enhancing my understanding of the novel's content and structure.