Natural Language Processing
AI-829

Nested Summarization with Heading Hierarchy

Mandate-2
Lexical Processing

Presented By:
Aprajita Kumari(MT2023037)
Nandini Yadav(MT2023096)

Contents

1. Introduction

The lexical processing phase of our NLP project is a critical step in preparing textual data for analysis and modeling. It involves a series of procedures aimed at refining raw text into a format that is more suitable for computational analysis. Through this phase, we aim to ensure that the text is standardized, cleaned, and organized, laying the groundwork for subsequent machine learning tasks.

2. Loading Data

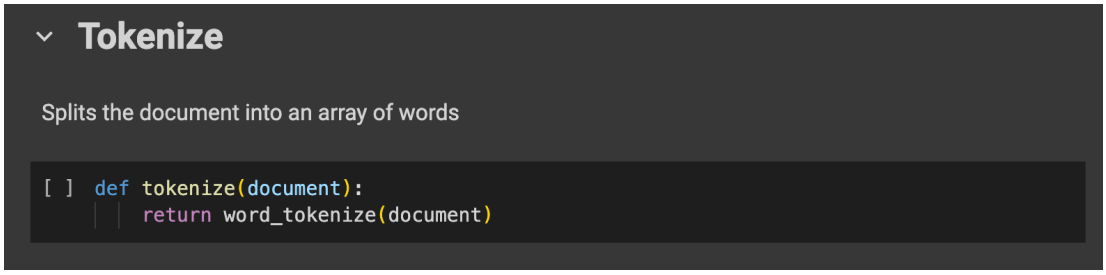
Initially, we retrieve the data from a designated directory containing textual documents.A dataset of news stories in CNN and Daily Mail popularly used for text summarization.These documents consist of news articles or stories along with their corresponding highlights. Upon loading, each document is parsed to segregate the main story content from its highlights, enabling us to work with these components separately for further processing.

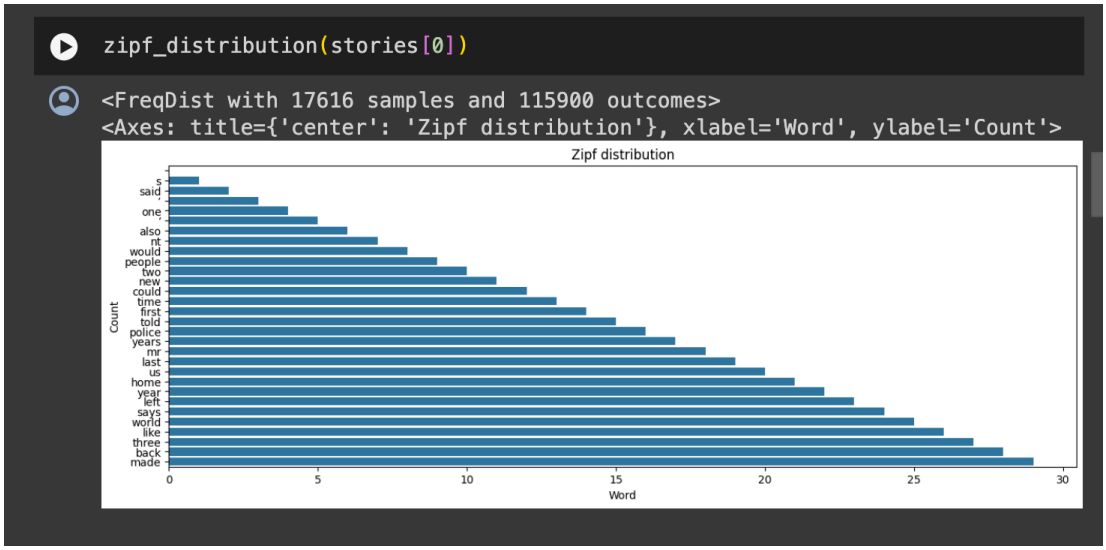
3. Lexical Processing

3.1 Tokenization and Zipf Distribution:

Tokenization, the process of breaking down text into individual tokens or words, is the first step in lexical processing.

Once tokenized, we then plotted a Zipf distribution to analyze the frequency distribution of words in the corpus.By removing extraneous elements such as punctuation and stopwords (commonly occurring words like "the", "and", "is"), we focus solely on content words, shedding light on their distribution and importance within the corpus.





```
[ ]  import matplotlib.pyplot as plt
     import seaborn as sns
     import numpy as np
```

```
     def remove_punctuation(text):
         punctuationfree= "".join([i for i in text if i not in punctuation])
         return punctuationfree
```

```
[ ]  def zipf_distribution(document):
         words = word_tokenize(document)
         filtered_words = []
         for word in words:
             if word.lower() not in stopwords:
                 if word.lower() not in punctuation:
                     filtered_words.append((remove_punctuation(word)).lower())

         word_frequencies = FreqDist(filtered_words)
         print(word_frequencies)
         labels = [element[0] for element in word_frequencies.most_common(30)]
         counts = [element[1] for element in word_frequencies.most_common(30)]
         plt.figure(figsize=(15,5))
         plt.title("Zipf distribution")
         plt.ylabel("Count")
         plt.xlabel("Word")
         plot = sns.barplot(labels)
         return plot
```

```
[ ]  zipf_distribution(stories[0])
```

3.2 Preprocessing Techniques:

In this step, we applied various techniques to clean the text data. Firstly, we removed extraneous elements such as URLs, HTML tags, brackets, and digits to ensure uniformity and eliminate irrelevant information.

Next, we expanded contractions to convert contracted forms into their expanded versions, enhancing readability. Contractions, like "don't" or "can't", are expanded into their full forms.

Additionally, we converted text to lowercase, removed punctuation, and eliminated stopwords to standardize the text format and create a cleaner, more manageable dataset.

Removing URL, HTML, brackets, digits

```
[ ]  def remove_url(data):
         return [re.sub(r'https://','', sentence) for sentence in data]
     def remove_html(data):
         return [BeautifulSoup(sentence, 'html.parser').get_text() for sentence in data]
     def remove_bracket(data):
         return [re.sub(r'[\([{})\]]','', sentence) for sentence in data]
     def remove_digit(data):
         return [re.sub('[0-9]','', sentence) for sentence in data]
     def remove_underscore(data):
         return [sentence.replace("_","") for sentence in data]
```

Expanding Contractions

Eg: he's -> he is

```
!pip install contractions
import contractions
```

```
Collecting contractions
  Downloading contractions-0.1.73-py2.py3-none-any.whl (8.7 kB)
Collecting textsearch>=0.0.21 (from contractions)
  Downloading textsearch-0.0.24-py2.py3-none-any.whl (7.6 kB)
Collecting anyascii (from textsearch>=0.0.21->contractions)
  Downloading anyascii-0.3.2-py3-none-any.whl (289 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 289.9/289.9 kB 5.9 MB/s eta 0:00:00
Collecting pyahocorasick (from textsearch>=0.0.21->contractions)
  Downloading pyahocorasick-2.0.0-cp310-cp310-manylinux_2_5_x86_64.manylinux1_x86_64.
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 110.8/110.8 kB 6.8 MB/s eta 0:00:00
Installing collected packages: pyahocorasick, anyascii, textsearch, contractions
Successfully installed anyascii-0.3.2 contractions-0.1.73 pyahocorasick-2.0.0 textsea
```

```
[ ]  def expand_contractions(sentence):
         contractions_expanded = [contractions.fix(word) for word in sentence.split()]
         return ' '.join(contractions_expanded)
```

Converting to lower case, removing punctuations and stopwords

```
def lower_case(tokens):
    return [word.lower() for word in tokens]
def remove_punctuation(tokens):
    return [re.sub(r'[^\w\s]', '', word) for word in tokens]
def remove_stopwords(tokens):
    return [word for word in tokens if word not in stopwords and word]
```

3.3 Canonicalization:

Canonicalization involves transforming words into their base or canonical forms. This step ensures that different variations of the same word are represented consistently, reducing redundancy and improving analysis accuracy. Within the lexical processing phase, we explore several canonicalization techniques, including:

3.3.1 Lemmatization:

Lemmatization involves deriving the base or root form of words based on their dictionary definitions.
Unlike stemming, which simply chops off prefixes or suffixes to approximate the root form, lemmatization considers the word's part of speech and grammar rules to provide more accurate results.
For example, the word "running" would be lemmatized to "run," while "better" would be lemmatized to “good."

To accomplish this, we performed part-of-speech tagging to determine the word's grammatical category, essential for accurate lemmatization.

By applying the WordNet Lemmatizer, we obtained the base form of each word based on its part of speech, ensuring consistency and interpretability in the text.

```
[ ]  nltk.download('wordnet')
     from nltk.stem import WordNetLemmatizer
     from nltk.corpus import wordnet

     [nltk_data] Downloading package wordnet to /root/nltk_data...
```

```
def pos_tags(tokenized):
    return nltk.tag.pos_tag(tokenized)

def get_wordnet_pos(tag):
    if tag.startswith('J'):
        return wordnet.ADJ
    elif tag.startswith('V'):
        return wordnet.VERB
    elif tag.startswith('N'):
        return wordnet.NOUN
    elif tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN

def to_wordnet(tokenized_pos):
    return [(word, get_wordnet_pos(pos_tag)) for (word, pos_tag) in tokenized_pos]
```

```
def lemmatizer(tokenized_stories):
    lemmatized_stories = []
    for story in tokenized_stories:
        tokenized_pos = pos_tags(story)
        tokenized_pos = to_wordnet(tokenized_pos)
        wnl = WordNetLemmatizer()
        lemmatized_stories.append([wnl.lemmatize(word, tag) for word, tag in tokenized_pos]
    return lemmatized_stories
```

3.3.2 Stemming:

Stemming is a simpler approach to canonicalization, where words are reduced to their root or stem form by removing prefixes or suffixes.
While stemming is less accurate than lemmatization, it is computationally less expensive and can still be effective in certain scenarios.
For instance, "running" would be stemmed to "run," and "cats" would be stemmed to “cat.”

```
from nltk.stem.porter import PorterStemmer
porter_stemmer = PorterStemmer()
```

```
def stemming(text):
    return [porter_stemmer.stem(word) for word in text]
```

3.3.3 Soundex:

Soundex is a phonetic algorithm used to encode words based on their pronunciation. This technique is particularly useful for tasks such as fuzzy matching or searching, where similar-sounding words need to be matched despite differences in spelling. By assigning a unique code to each word based on its sound, Soundex facilitates efficient retrieval and comparison of words with similar pronunciations.

```
def get_soundex(word):
    word = word.upper()
    soundex = ""
    soundex += word[0]
    dictionary = {"BFPV": "1", "CGJKQSXZ":"2", "DT":"3", "L":"4", "MN":"5"

    for char in word[1:]:
        for key in dictionary.keys():
            if char in key:
                code = dictionary[key]
                if code != soundex[-1]:
                    soundex += code

    soundex = soundex.replace(".", "")
    soundex = soundex[:4].ljust(4, "0")
    return soundex
```

3.3.4 Edit Distance:

Edit distance measures the similarity between two words by calculating the minimum number of operations (such as insertions, deletions, or substitutions) required to transform one word into another. This metric is valuable for tasks like spell checking or text correction, where identifying similar words with minor differences is crucial.

```
[ ] def get_edit_distance(w1,w2):
        return nltk.edit_distance(w1, w2, transpositions=False)
```

3.3.5 Combining into a Pipeline:

To streamline the preprocessing process, all the aforementioned techniques are integrated into a cohesive pipeline. This pipeline serves as a standardized workflow, ensuring that each document undergoes the same preprocessing steps consistently. By automating the process, we enhance efficiency and reduce the risk of human error, facilitating the handling of large volumes of text data.

In our lexical processing phase, we adopt a comprehensive approach to canonicalization, leveraging both lemmatization and stemming techniques to derive the base forms of words. We prioritize lemmatization for its accuracy, ensuring that words are transformed into their true root forms based on their part of speech.

Additionally, we explore Soundex encoding and edit distance calculations to facilitate tasks such as fuzzy matching and text correction, enhancing the versatility and robustness of our lexical processing pipeline.

Combining into a pipeline

```
[ ] def process(stories):
        stories = remove_url(stories)
        stories = remove_html(stories)
        stories = remove_bracket(stories)
        stories = remove_digit(stories)
        stories = remove_underscore(stories)

        processed_list = []
        for story in stories:
            processed = expand_contractions(story)
            processed = tokenize(processed)
            processed = lower_case(processed)
            processed = remove_punctuation(processed)
            processed = remove_stopwords(processed)
            processed_list.append(processed)

        return lemmatizer(processed_list)
```

```
processed_stories = process(stories[:10])
processed_stories
```

```
  'publish',
  'est',
  'october',
  'updated',
  'est',
  'october',
  'bishop',
  'fargo',
  'catholic',
  'diocese',
  'north',
  'dakota',
  'expose',
  'potentially',
  'hundred',
  'church',
  'member',
  'fargo',
  'grand',
  'fork',
  'jamestown',
  'hepatitis',
  'virus',
  'late',
  'september',
  'early',
  'october',
  'state',
  'health',
  'department',
  'issue',
```

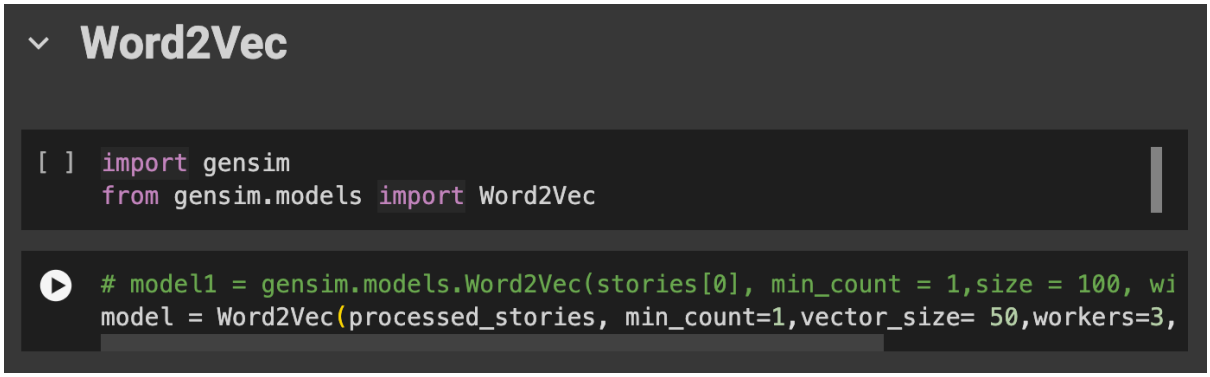
By incorporating these canonicalization techniques into our workflow, we strive to enhance the quality and consistency of the textual data, ultimately empowering our machine learning algorithms to extract meaningful insights and patterns with greater accuracy and efficiency.

3.4. Word2Vec

With the preprocessed text in hand, we proceed to generate word embeddings using the Word2Vec model. Word2Vec transforms words into dense vector representations, capturing semantic similarities and relationships between them. By training the Word2Vec model on our preprocessed text, we aim to encode semantic information into numerical vectors, enabling downstream machine learning tasks to operate on a more meaningful and interpretable level.

During training, the Word2Vec model learns to predict the context words surrounding a target word within a given window size. By analyzing the co-occurrence patterns of words in the text corpus, the model gradually learns to encode semantic similarities and relationships between words into the vector space.

Once trained, the Word2Vec model generates dense vector embeddings for each word in the vocabulary, encapsulating semantic information such as word meanings, context, and relationships. These embeddings serve as powerful numerical representations of the textual data, enabling downstream machine learning algorithms to operate on a more meaningful and interpretable level.



4.Conclusion

Tokenization and Zipf distribution analysis allowed us to understand the distribution of words in the corpus, guiding further preprocessing steps to enhance the quality of the text data.

Preprocessing techniques ensured that the text data was clean, consistent, and free from noise, preparing it for accurate interpretation by machine learning algorithms.

Canonicalization techniques like lemmatization and stemming enhanced the consistency and interpretability of the text, facilitating more accurate analysis and understanding.

Integration of Word2Vec facilitated the transformation of textual data into meaningful numerical representations, enabling downstream machine learning tasks to operate on a more meaningful and interpretable level.

5. Collab Notebook

https://colab.research.google.com/drive/1Ur2yWgCLP7AmJubepPrPqya32MELGb2A?usp=sharing