Lab 1. Introduction to Natural Language Processing

Overview

In this lab, you will learn the difference between **Natural Language Processing** (NLP) and basic text analytics. You will implement various preprocessing tasks such as tokenization, lemmatization, stemming, stop word removal, and more. By the end of this lab, you will have a deep understanding of the various phases of an NLP project, from data collection to model deployment.

Introduction

NLP works at different levels, which means that machines process and understand natural language at different levels. These levels are as follows:

- **Morphological level**: This level deals with understanding word structure and word information.
- **Lexical level**: This level deals with understanding the part of speech of the word.
- **Syntactic level**: This level deals with understanding the syntactic analysis of a sentence, or parsing a sentence.
- **Semantic level**: This level deals with understanding the actual meaning of a sentence.
- **Discourse level**: This level deals with understanding the meaning of a sentence beyond just the sentence level, that is, considering the context.
- **Pragmatic level**: This level deals with using real-world knowledge to understand the sentence.

Text Analytics and NLP

Text analytics is the method of extracting meaningful insights and answering questions from text data, such as those to do with the length of sentences, length of words, word count, and finding words from the text.

NLP helps us in understanding the semantics and the underlying meaning of text. It can be broadly categorized into two types: **Natural Language Understanding** (**NLU**) and **Natural Language Generation** (**NLG**). A proper explanation of these terms is provided here:

- **NLU**: NLU refers to a process by which an inanimate object with computing power is able to comprehend spoken language.
- **NLG**: NLG refers to a process by which an inanimate object with computing power is able to communicate with humans in a language that they can understand.

Exercise 1.01: Basic Text Analytics

In this exercise, we will perform some basic text analytics on some given text data, including searching for a particular word, finding the index of a word, and finding a word at a given position. Follow these steps to implement this exercise using the following sentence:

"The quick brown fox jumps over the lazy dog."

1. Open a Jupyter Notebook.
2. Assign a `sentence` variable the value `'The quick brown fox jumps over the lazy dog'`. Insert a new cell and add the following code to implement this:

```
sentence = 'The quick brown fox jumps over the lazy dog'
sentence
```

3. Check whether the word `'quick'` belongs to that text using the following code:

```
def find_word(word, sentence):
    return word in sentence
find_word('quick', sentence)
```

   The preceding code will return the output `'True'`.

4. Find out the `index` value of the word `'fox'` using the following code:

```
def get_index(word, text):
    return text.index(word)
get_index('fox', sentence)
```

   The code will return the output `16`.

5. To find out the rank of the word `'lazy'`, use the following code:

```
get_index('lazy', sentence.split())
```

   This code generates the output `7`.

6. To print the third word of the given text, use the following code:

```
def get_word(text,rank):
    return text.split()[rank]
get_word(sentence,2)
```

   This will return the output `brown`.

7. To print the third word of the given sentence in reverse order, use the following code:

```
get_word(sentence,2)[::-1]
```

   This will return the output `nworb`.

8. To concatenate the first and last words of the given sentence, use the following code:

```
def concat_words(text):
    """
    This method will concat first and last
    words of given text
    """
    words = text.split()
    first_word = words[0]
    last_word = words[len(words)-1]
    return first_word + last_word
concat_words(sentence)
```

Note:

The triple-quotes (`"""`) shown in the code snippet above are used to denote the start and end points of a multi-line code comment. Comments are added into code to help explain specific bits of logic.

The code will generate the output `Thedog`.

9. To print words at even positions, use the following code:

```
def get_even_position_words(text):
    words = text.split()
    return [words[i] for i in range(len(words)) if i%2 == 0]
get_even_position_words(sentence)
```

This code generates the following output:

```
['The', 'brown', 'jumps', 'the', 'dog']
```

10. To print the last three letters of the text, use the following code:

```
def get_last_n_letters(text, n):
    return text[-n:]
get_last_n_letters(sentence,3)
```

This will generate the output `dog`.

11. To print the text in reverse order, use the following code:

```
def get_reverse(text):
    return text[::-1]
get_reverse(sentence)
```

This code generates the following output:

```
'god yzal eht revo spmuj xof nworb kciuq ehT'
```

12. To print each word of the given text in reverse order, maintaining their sequence, use the following code:

```
def get_word_reverse(text):
    words = text.split()
    return ' '.join([word[::-1] for word in words])
get_word_reverse(sentence)
```

This code generates the following output:

```
ehT kciuq nworb xof spmuj revo eht yzal god
```

We are now well acquainted with basic text analytics techniques.

Note In the next section, let's dive deeper into the various steps and subtasks in NLP.

Various Steps in NLP

Tokenization

NLTK provides a method called `word_tokenize()`, which tokenizes given text into words. It actually separates the text into different words based on punctuation and spaces between words.

To get a better understanding of tokenization, let's solve an exercise based on it in the next section.

Exercise 1.02: Tokenization of a Simple Sentence

In this exercise, we will tokenize the words in a given sentence with the help of the **NLTK** library. Follow these steps to implement this exercise using the sentence, "I am reading NLP Concepts."

1. Open a Jupyter Notebook.
2. Insert a new cell and add the following code to import the necessary libraries and download the different types of NLTK data that we are going to use for different tasks in the following exercises:

   ```
   from nltk import word_tokenize, download
   download(['punkt','averaged_perceptron_tagger','stopwords'])
   ```

   In the preceding code, we are using NLTK's `download()` method, which downloads the given data from NLTK. NLTK data contains different corpora and trained models. In the preceding example, we will be downloading the stop word list, `'punkt'`, and a perceptron tagger, which is used to implement parts of speech tagging using a structured algorithm. The data will be downloaded at `nltk_data/corpora/` in the home directory of your computer. Then, it will be loaded from the same path in further steps.
3. The `word_tokenize()` method is used to split the sentence into words/tokens. We need to add a sentence as input to the `word_tokenize()` method so that it performs its job. The result obtained will be a list, which we will store in a `word` variable. To implement this, insert a new cell and add the following code:

```
```
def get_tokens(sentence):
    words = word_tokenize(sentence)
    return words
```
```

4. In order to view the list of tokens generated, we need to view it using the `print()` function. Insert a new cell and add the following code to implement this:

   ```
   print(get_tokens("I am reading NLP Concepts."))
   ```

   This code generates the following output:

   ```
   ['I', 'am', 'reading', 'NLP', 'Fundamentals', '.']
   ```

We can see the list of tokens generated with the help of the `word_tokenize()` method.

Note In the next section, we will see another pre-processing step: **Parts-of-Speech (PoS) tagging**.

PoS Tagging

In NLP, the term PoS refers to parts of speech. PoS tagging refers to the process of tagging words within sentences with their respective PoS. We extract the PoS of tokens constituting a sentence so that we can filter out the PoS that are of interest and analyze them. For example, if we look at the sentence, "The sky is blue," we get four tokens, namely "The," "sky," "is," and "blue", with the help of tokenization. Now, using a **PoS tagger**, we tag the PoS for each word/token. This will look as follows:

```
[('The', 'DT'), ('sky', 'NN'), ('is', 'VBZ'), ('blue', 'JJ')]
```

The preceding format is an output of the NLTK `pos_tag()` method. It is a list of tuples in which every tuple consists of the word followed by the PoS tag:

`DT` = Determiner

`NN` = Noun, common, singular or mass

`VBZ` = Verb, present tense, third-person singular

`JJ` = Adjective

Let's perform a simple exercise to understand how PoS tagging is done in Python.

Exercise 1.03: PoS Tagging

In this exercise, we will find out the PoS for each word in the sentence, `I am reading NLP Concepts`. We first make use of tokenization in order to get the tokens. Later, we will use the `pos_tag()` method, which will help us find the PoS for each word/token. Follow these steps to implement this exercise:

1. Open a Jupyter Notebook.
2. Insert a new cell and add the following code to import the necessary libraries:

```
```
from nltk import word_tokenize, pos_tag
```
```

3. To find the tokens in the sentence, we make use of the `word_tokenize()` method. Insert a new cell and add the following code to implement this:

```
```
def get_tokens(sentence):
    words = word_tokenize(sentence)
    return words
```
```

4. Print the tokens with the help of the `print()` function. To implement this, add a new cell and write the following code:

   ```
   words  = get_tokens("I am reading NLP Concepts")
   print(words)
   ```

   This code generates the following output:

   ```
   ['I', 'am', 'reading', 'NLP', 'Fundamentals']
   ```

5. We'll now use the `pos_tag()` method. Insert a new cell and add the following code:

```
def get_pos(words):
    return pos_tag(words)
get_pos(words)
```

This code generates the following output:

```
[('I', 'PRP'),
 ('am', 'VBP'),
 ('reading', 'VBG'),
 ('NLP', 'NNP'),
 ('Fundamentals', 'NNS')]
```

In the preceding output, we can see that for each token, a PoS has been allotted. Here, **PRP** stands for **personal pronoun**, **VBP** stands for **verb present**, **VGB** stands for **verb gerund**, **NNP** stands for **proper noun singular**, and **NNS** stands for **noun plural**.

Note

We have learned about assigning appropriate PoS labels to tokens in a sentence. In the next section, we will learn about **stop words** in sentences and ways to deal with them.

Stop Word Removal

Stop words are the most frequently occurring words in any language and they are just used to support the construction of sentences and do not contribute anything to the semantics of a sentence. Examples of stop words include "a," "am," "and," "the," "in," "of," and more.

In the next exercise, we will look at the practical implementation of removing stop words from a given sentence.

Exercise 1.04: Stop Word Removal

In this exercise, we will check the list of stop words provided by the `nltk` library. Based on this list, we will filter out the stop words included in our text:

1. Open a Jupyter Notebook.
2. Insert a new cell and add the following code to import the necessary libraries:

```
```
from nltk import download
download('stopwords')
from nltk import word_tokenize
from nltk.corpus import stopwords
```
```

3. In order to check the list of stop words provided for `English`, we pass it as a parameter to the `words()` function. Insert a new cell and add the following code to implement this:

```
```
stop_words = stopwords.words('english')
```
```

4. In the code, the list of stop words provided by `English` is stored in the `stop_words` variable. In order to view the list, we make use of the `print()` function. Insert a new cell and add the following code to view the list:

```
print(stop_words)
```

This code generates the following output:

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've",
"you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himsel
f', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'the
m', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "tha
t'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have',
'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'i
f', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'ag
ainst', 'between', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to',
'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'the
n', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each',
'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same',
'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "s
hould've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn',
"couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'hav
en', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn',
"needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't",
'won', "won't", 'wouldn', "wouldn't"]
```

5. To remove the stop words from a sentence, we first assign a string to the `sentence` variable and tokenize it into words using the `word_tokenize()` method. Insert a new cell and add the following code to implement this:

```
sentence = "I am learning Python. It is one of the "\
           "most popular programming languages"
sentence_words = word_tokenize(sentence)
```

Note:

The code snippet shown here uses a backslash (`\`) to split the logic across multiple lines. When the code is executed, Python will ignore the backslash, and treat the code on the next line as a direct continuation of the current line.

6. To print the list of tokens, insert a new cell and add the following code:

```
print(sentence_words)
```

This code generates the following output:

```
['I', 'am', 'learning', 'Python', '.', 'It', 'is', 'one', 'of', 'the', 'most',
'popular', 'programming', 'languages']
```

7. To remove the stop words, we need to loop through each word in the sentence, check whether there are any stop words, and then finally combine them to form a complete sentence. To implement this, insert a new cell and add the following code:

```
```
def remove_stop_words(sentence_words, stop_words):
    return ' '.join([word for word in sentence_words if \
```
```

```
                        word not in stop_words])
```

8. To check whether the stop words are filtered out from our sentence, print the `sentence_no_stops` variable. Insert a new cell and add the following code to print:

   ```
   print(remove_stop_words(sentence_words,stop_words))
   ```

   This code generates the following output:

   ```
   I learning Python. It one popular programming languages
   ```

   As you can see in the preceding code snippet, stop words such as "am," "is," "of," "the," and "most" are being filtered out and text without stop words is produced as output.

9. Add your own stop words to the stop word list:

   ```
   stop_words.extend(['I','It', 'one'])
   print(remove_stop_words(sentence_words,stop_words))
   ```

   This code generates the following output:

   ```
   learning Python . popular programming languages
   ```

As we can see from the output, now words such as "I," "It," and* "One" are removed as we have added them to our custom stop word list. We have learned how to remove stop words from given text.

Text Normalization

We need to perform text normalization as there are some words that can mean the same thing as each other. There are various ways of normalizing text, such as spelling correction, stemming, and lemmatization, which will be covered later.

Exercise 1.05: Text Normalization

In this exercise, we will normalize some given text. Basically, we will be trying to replace select words with new words, using the `replace()` function, and finally produce the normalized text. `replace()` is a built-in Python function that works on strings and takes two arguments. It will return a copy of a string in which the occurrence of the first argument will be replaced by the second argument.

Follow these steps to complete this exercise:

1. Open a Jupyter Notebook.
2. Insert a new cell and add the following code to assign a string to the `sentence` variable:

```
```
sentence = "I visited the US from the UK on 22-10-18"
```
```

3. We want to replace `"US"` with `"United States"`, `"UK"` with `"United Kingdom"`, and `"18"` with `"2018"`. To do so, use the `replace()` function and store the updated output in the `"normalized_sentence"` variable. Insert a new cell and add the following code to implement this:

```
def normalize(text):
    return text.replace("US", "United States")\
               .replace("UK", "United Kingdom")\
               .replace("-18", "-2018")
```

4. To check whether the text has been normalized, insert a new cell and add the following code to print it:

```
normalized_sentence = normalize(sentence)
print(normalized_sentence)
```

The code generates the following output:

```
I visited the United States from the United Kingdom on 22-10-2018
```

5. Add the following code:

```
normalized_sentence = normalize('US and UK are two superpowers')
print(normalized_sentence)
```

The code generates following output:

```
United States and United Kingdom are two superpowers
```

In the preceding code, we can see that our text has been normalized.

Note Over the next sections, we will explore various other ways in which text can be normalized.

Spelling Correction

Spelling correction is one of the most important tasks in any NLP project. It can be time-consuming, but without it, there are high chances of losing out on important information.

Spelling correction is executed in two steps:

1. Identify the misspelled word, which can be done by a simple dictionary lookup.
2. Replace it or suggest the correctly spelled word.

We make use of the `autocorrect` Python library to correct spellings.

`autocorrect` is a Python library used to correct the spelling of misspelled words for different languages. It provides a method called `spell()`, which takes a word as input and returns the correct spelling of the word.

Let's look at the following exercise to get a better understanding of this.

Exercise 1.06: Spelling Correction of a Word and a Sentence

In this exercise, we will perform spelling correction on a word and a sentence, with the help of Python's `autocorrect` library. Follow these steps in order to complete this exercise:

1. Open a Jupyter Notebook.
2. Insert a new cell and add the following code to import the necessary libraries:

```
from nltk import word_tokenize
from autocorrect import Speller
```

3. In order to correct the spelling of a word, pass a wrongly spelled word as a parameter to the `spell()` function. Before that, you have to create a `spell` object of the `Speller` class using `lang='en'` to signify the English language. Insert a new cell and add the following code to implement this:

   ```
   spell = Speller(lang='en')
   spell('Natureal')
   ```

   This code generates the following output:

   ```
   'Natural'
   ```

4. To correct the spelling of a sentence, first tokenize it into tokens. After that, loop through each token in `sentence`, autocorrect the words, and finally combine the words. Insert a new cell and add the following code to implement this:

```
sentence = word_tokenize("Ntural Luanguage Processin deals with "\
                         "the art of extracting insightes from "\
                         "Natural Languaes")
```

5. Use the `print()` function to print all tokens. Insert a new cell and add the following code to print the tokens:

   ```
   print(sentence)
   ```

   This code generates the following output:

   ```
   ['Ntural', 'Luanguage', 'Processin', 'deals', 'with', 'the', 'art', 'of',
   'extracting', 'insightes', 'from', 'Natural', 'Languaes']
   ```

6. Now that we have got the tokens, loop through each token in `sentence`, correct the tokens, and assign them to a new variable. Insert a new cell and add the following code to implement this:

```
def correct_spelling(tokens):
    sentence_corrected = ' '.join([spell(word) \
                                   for word in tokens])
    return sentence_corrected
```

7. To print the correct sentence, insert a new cell and add the following code:

   ```
   print(correct_spelling(sentence))
   ```

This code generates the following output:

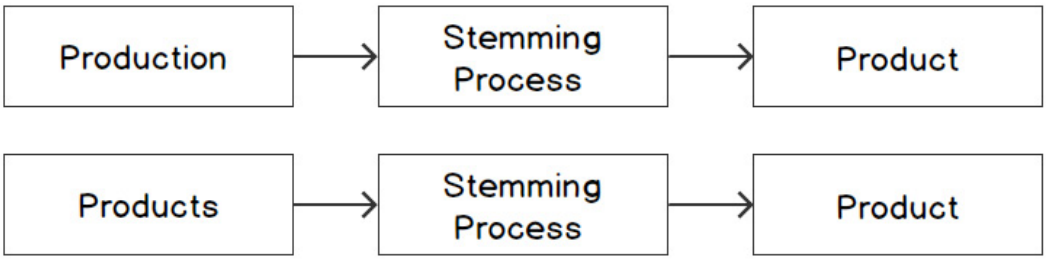
```
['Natural', 'Language', 'Procession', 'deals', 'with', 'the', 'art',
 'of', 'extracting', 'insights', 'from', 'Natural', 'Languages']
```

In the preceding code snippet, we can see that most of the wrongly spelled words have been corrected. But the word "`Processin`" was wrongly converted into "`Procession`." It should have been "`Processing`." This happened because to change "`Processin`" to "`Procession`" or "`Processing`," an equal number of edits is required. To rectify this, we need to use other kinds of spelling correctors that are aware of context.

Note In the next section, we will look at stemming, which is another form of text normalization.

Stemming

In most languages, words get transformed into various forms when being used in a sentence. For example, the word "product" might get transformed into "production" when referring to the process of making something or transformed into "products" in plural form. It is necessary to convert these words into their base forms, as they carry the same meaning in any case. Stemming is the process that helps us to do so. If we look at the following figure, we get a perfect idea of how words get transformed into their base forms:



To get a better understanding of stemming, let's perform a simple exercise.

In this exercise, we will be using two algorithms, called the porter stemmer and the snowball stemmer, provided by the NLTK library. The porter stemmer is a rule-based algorithm that transforms words to their base form by removing suffixes from words. The snowball stemmer is an improvement over the porter stemmer and is a little bit faster and uses less memory. In NLTK, this is done by the `stem()` method provided by the `PorterStemmer` class.

Exercise 1.07: Using Stemming

In this exercise, we will pass a few words through the stemming process so that they get converted into their base forms. Follow these steps to implement this exercise:

1. Open a Jupyter Notebook.
2. Insert a new cell and add the following code to import the necessary libraries:

```
```
from nltk import stem
```
```

3. Now pass the following words as parameters to the `stem()` method. To implement this, insert a new cell and add the following code:

```
def get_stems(word,stemmer):
    return stemmer.stem(word)
porterStem = stem.PorterStemmer()
get_stems("production",porterStem)
```

4. When the input is `"production"`, the following output is generated:

```
'product'
```

5. Similarly, the following code would be used for the input `"coming"`.

   ```
   get_stems("coming",porterStem)
   ```

   We get the following output:

   ```
   'come'
   ```

6. Similarly, the following code would be used for the input `"firing"`.

   ```
     get_stems("firing",porterStem)
   ```

   When the input is `"firing"`, the following output is generated:

   ```
   'fire'
   ```

7. The following code would be used for the input `"battling"`.

   ```
     get_stems("battling",porterStem)
   ```

   If we give the input `"battling"`, the following output is generated:

   ```
   'battl'
   ```

8. The following code will also generate the same output as above, for the input `"battling".`

   ```
   stemmer = stem.SnowballStemmer("english")
   get_stems("battling",stemmer)
   ```

   The output will be as follows:

   ```
   'battl'
   ```

As you have seen while using the snowball stemmer, we have to provide the language as `"english"`. We can also use the stemmer for different languages such as Spanish, French, and many more. From the preceding code snippets, we can see that the entered words are converted into their base forms.

Note In the next section, we will focus on **lemmatization**, which is another form of text normalization.

Lemmatization

Sometimes, the stemming process leads to incorrect results. For example, in the last exercise, the word `battling` was transformed to `"battl"`, which is not a word. To overcome such problems with stemming, we make use of lemmatization. Lemmatization is the process of converting words to their base grammatical form, as in "battling" to "battle," rather than just randomly axing words. In this process, an additional check is made by looking through a dictionary to extract the base form of a word. Getting more accurate results requires some additional information; for example, PoS tags along with words will help in getting better results.

In the following exercise, we will be using `WordNetLemmatizer`, which is an NLTK interface of WordNet. WordNet is a freely available lexical English database that can be used to generate semantic relationships between words. NLTK's `WordNetLemmatizer` provides a method called `lemmatize()`, which returns the lemma (grammatical base form) of a given word using WordNet.

To put lemmatization into practice, let's perform an exercise where we'll use the `lemmatize()` function.

Exercise 1.08: Extracting the Base Word Using Lemmatization

In this exercise, we will use the lemmatization process to produce the proper form of a given word. Follow these steps to implement this exercise:

1. Open a Jupyter Notebook.
2. Insert a new cell and add the following code to import the necessary libraries:

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

3. Create an object of the `WordNetLemmatizer` class. Insert a new cell and add the following code to implement this:

```
```
lemmatizer = WordNetLemmatizer()
```
```

4. Bring the word to its proper form by using the `lemmatize()` method of the `WordNetLemmatizer` class. Insert a new cell and add the following code to implement this:

   ```
   def get_lemma(word):
       return lemmatizer.lemmatize(word)
   get_lemma('products')
   ```

   With the input `products`, the following output is generated:

   ```
   'product'
   ```

5. Similarly, use the input as `production` now:

   ```
   get_lemma('production')
   ```

With the input `production`, the following output is generated:

```
'production'
```

6. Similarly, use the input as `coming` now:

```
get_lemma('coming')
```

With the input `coming`, the following output is generated:

```
'coming'
```

Hence, we have learned how to use the lemmatization process to transform a given word into its base form.

Note: In the next section, we will look at another preprocessing step in NLP:

named entity recognition (**NER**).

Named Entity Recognition (NER)

NER is the process of extracting important entities, such as person names, place names, and organization names, from some given text. These are usually not present in dictionaries. So, we need to treat them differently. The main objective of this process is to identify the named entities (such as proper nouns) and map them to categories, which are already defined. For example, categories might include names of people, places, and so on.

NER has found use in many NLP tasks, including assigning tags to news articles, search algorithms, and more. NER can analyze a news article and extract the major people, organizations, and places discussed in it and assign them as tags for new articles.

In the case of search algorithms, let's suppose we have to create a search engine, meant specifically for books. If we were to submit a given query for all the words, the search would take a lot of time. Instead, if we extract the top entities from all the books using NER and run a search query on the entities rather than all the content, the speed of the system would increase dramatically.

To get a better understanding of this process, we'll perform an exercise. Before moving on to the exercise, let me introduce you to chunking, which we are going to use in the following exercise. Chunking is the process of grouping words together into chunks, which can be further used to find noun groups and verb groups, or can also be used for sentence partitioning.

Exercise 1.09: Treating Named Entities

In this exercise, we will find the named entities in a given sentence. Follow these steps to implement this exercise using the following sentence:

"We are reading a book published by Fenago which is based out of Birmingham."

1. Open a Jupyter Notebook.
2. Insert a new cell and add the following code to import the necessary libraries:

```
```
from nltk import download
from nltk import pos_tag
from nltk import ne_chunk
```
```

```
from nltk import word_tokenize
download('maxent_ne_chunker')
download('words')
```

3. Declare the `sentence` variable and assign it a string. Insert a new cell and add the following code to implement this:

```
sentence = "We are reading a book published by Fenago "\
           "which is based out of Birmingham."
```

4. To find the named entities from the preceding text, insert a new cell and add the following code:

```
def get_ner(text):
    i = ne_chunk(pos_tag(word_tokenize(text)), binary=True)
    return [a for a in i if len(a)==1]
get_ner(sentence)
```

This code generates the following output:

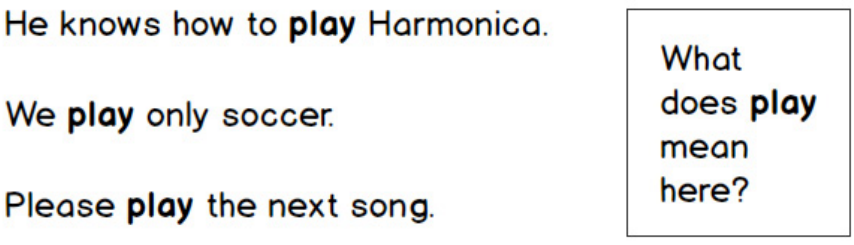
```
[Tree('NE', [('Fenago', 'NNP')]), Tree('NE', [('Birmingham', 'NNP')])]
```

In the preceding code, we can see that the code identifies the named entities "`Fenago`" and "`Birmingham`" and maps them to an already-defined category, "`NNP`."

Note: In the next section, we will focus on word sense disambiguation, which helps us to identify the right sense of any word.

Word Sense Disambiguation

There's a popular saying: "A man is known by the company he keeps.'' Similarly, a word's meaning depends on its association with other words in a sentence. This means two or more words with the same spelling may have different meanings in different contexts. This often leads to ambiguity. Word sense disambiguation is the process of mapping a word to the sense that it should carry. We need to disambiguate words based on the sense they carry so that they can be treated as different entities when being analyzed. The following figure displays a perfect example of how ambiguity is caused due to the usage of the same word in different sentences:



One of the algorithms to solve word sense disambiguation is the Lesk algorithm. It has a huge corpus in the background (generally `WordNet` is used) that contains definitions of all the possible synonyms of all the possible words in a language. Then it takes a word and the context as input and finds a match between the context and all the

definitions of the word. The meaning with the highest number of matches with the context of the word will be returned.

For example, suppose we have a sentence such as "We play only soccer" in a given text. Now, we need to find the meaning of the word "play" in this sentence. In the Lesk algorithm, each word with ambiguous meaning is saved in background synsets. In this case, the word "play" will be saved with all possible definitions. Let's say we have two definitions of the word "play":

1. Play: Participating in a sport or game
2. Play: Using a musical instrument

Then, we will find the similarity between the context of the word "play" in the text and both of the preceding definitions using text similarity techniques. The definition best suited to the context of "play" in the sentence will be considered the meaning or definition of the word. In this case, we will find that our first definition fits best in context, as the words "sport" and "game" are present in the preceding sentences.

In the next exercise, we will be using the Lesk module from NLTK. It takes a sentence and the word as input, and returns the meaning or definition of the word. The output of the Lesk method is `synset`, which contains the ID of the matched definition. These IDs can be matched with their definitions using the `definition()` method of `wsd.synset('word')`.

To get a better understanding of this process, let's look at an exercise.

Exercise 1.10: Word Sense Disambiguation

In this exercise, we will find the sense of the word "bank" in two different sentences. Follow these steps to implement this exercise:

1. Open a Jupyter Notebook.
2. Insert a new cell and add the following code to import the necessary libraries:

```
```
import nltk
nltk.download('wordnet')
from nltk.wsd import lesk
from nltk import word_tokenize
```
```

3. Declare two variables, `sentence1` and `sentence2`, and assign them with appropriate strings. Insert a new cell and the following code to implement this:

```
```
sentence1 = "Keep your savings in the bank"
sentence2 = "It's so risky to drive over the banks of the road"
```
```

4. To find the sense of the word "bank" in the preceding two sentences, use the Lesk algorithm provided by the `nltk.wsd` library. Insert a new cell and add the following code to implement this:

```
def get_synset(sentence, word):
    return lesk(word_tokenize(sentence), word)
get_synset(sentence1,'bank')
```

This code generates the following output:

```
Synset('savings_bank.n.02')
```

5. Here, `savings_bank.n.02` refers to a container for keeping money safely at home. To check the other sense of the word "bank," write the following code:

   ```
   get_synset(sentence2,'bank')
   ```

   This code generates the following output:

   ```
   Synset('bank.v.07')
   ```

   Here, `bank.v.07` refers to a slope in the turn of a road.

   Thus, with the help of the Lesk algorithm, we were able to identify the sense of a word in whatever context.

   Note: In the next section, we will focus on **sentence boundary detection**, which helps detect the start and end points of sentences.

Sentence Boundary Detection

Sentence boundary detection is the method of detecting where one sentence ends and another begins. If you are thinking that this sounds pretty easy, as a period (.) or a question mark (?) denotes the end of a sentence and the beginning of another sentence, then you are wrong. There can also be instances where the letters of acronyms are separated by full stops, for instance. Various analyses need to be performed at a sentence level; detecting the boundaries of sentences is essential.

An exercise will provide us with a better understanding of this process.

Exercise 1.11: Sentence Boundary Detection

In this exercise, we will extract sentences from a paragraph. To do so, we'll be using the `sent_tokenize()` method, which is used to detect sentence boundaries. The following steps need to be performed:

1. Open a Jupyter Notebook.
2. Insert a new cell and add the following code to import the necessary libraries:

```
```
import nltk
from nltk.tokenize import sent_tokenize
```
```

3. Use the `sent_tokenize()` method to detect sentences in some given text. Insert a new cell and add the following code to implement this:

   ```
   def get_sentences(text):
       return sent_tokenize(text)
   get_sentences("We are reading a book. Do you know who is "\
                 "the publisher? It is Fenago. Fenago is based "\
                 "out of Birmingham.")
   ```

This code generates the following output:

```
['We are reading a book.'
 'Do you know who is the publisher?'
 'It is Fenago.',
 'Fenago is based out of Birmingham.']
```

4. Use the `sent_tokenize()` method for text that contains periods (.) other than those found at the ends of sentences:

```
get_sentences("Mr. Donald John Trump is the current "\
              "president of the USA. Before joining "\
              "politics, he was a businessman.")
```

The code will generate the following output:

```
['Mr. Donald John Trump is the current president of the USA.',
 'Before joining politics, he was a businessman.']
```

As you can see in the code, the `sent_tokenize` method is able to differentiate between the period (.) after "Mr" and the one used to end the sentence. We have covered all the preprocessing steps that are involved in NLP.

Activity 1.01: Preprocessing of Raw Text

We have a text corpus that is in an improper format. In this activity, we will perform all the preprocessing steps that were discussed earlier to get some meaning out of the text.

Note

The text corpus, `file.txt`, can be found at this location: https://github.com/fenago/nlp-generative-ai-bootcamp/blob/master/Lab01/data/file.txt

Follow these steps to implement this activity:

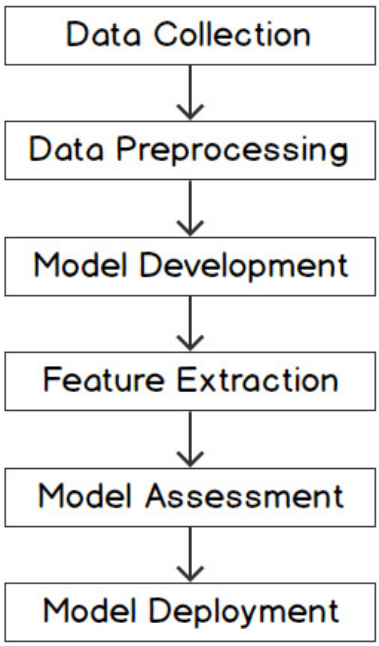
1. Import the necessary libraries.
2. Load the text corpus to a variable.
3. Apply the tokenization process to the text corpus and print the first 20 tokens.
4. Apply spelling correction on each token and print the initial 20 corrected tokens as well as the corrected text corpus.
5. Apply PoS tags to each of the corrected tokens and print them.
6. Remove stop words from the corrected token list and print the initial 20 tokens.
7. Apply stemming and lemmatization to the corrected token list and then print the initial 20 tokens.
8. Detect the sentence boundaries in the given text corpus and print the total number of sentences.

   Note: The solution to this activity in the current directory.

We have learned about and achieved the preprocessing of given data. By now, you should be familiar with what NLP is and what basic preprocessing steps are needed to carry out any NLP project. In the next section, we will focus on the different phases of an NLP project.

Kick Starting an NLP Project

We can divide an NLP project into several sub-projects or phases. These phases are completed in a particular sequence. This tends to increase the overall efficiency of the process, as memory usage changes from one phase to the next. An NLP project has to go through six major phases, which are outlined in the following figure:



Summary

In this lab, we learned about the basics of NLP and how it differs from text analytics. We covered the various preprocessing steps that are included in NLP, such as tokenization, PoS tagging, stemming, lemmatization, and more. We also looked at the different phases an NLP project has to pass through, from data collection to model deployment.

In the next lab, you will learn about the different methods of extracting features from unstructured text, such as TF-IDF and bag of words. You will also learn about NLP tasks such as tokenization, lemmatization, and stemming in more detail. Furthermore, text visualization techniques such as word clouds will be introduced.