

1. Lab#01 - Introduction to Natural Language processing (NLTK)

Learning Objectives:

- Implementation of NLP models using NLTK.
- Implementation of POS-tagging from scratch.
- Implementation of Stopword library by scratch.
- Implementation of stemming lemmatization of corpus.

Outcomes:

- Students should be able to implement NLP models.
 - Students should be able to create their own list of dictionaries for stop words and stemming.
 - Students should be able to Preprocess the data.s
-

1.1 Why should I learn NLP?

AI is rapidly penetrating various facets of our lives, from being our home assistant to fielding our queries as automated tech support. Various industry outlook reports project that AI will create millions of jobs (projection range between 200 and 500 million) worldwide by the year 2030. The majority of these jobs will require ML and NLP skills, and therefore it is imperative for engineers and technologists to upskill and prepare for the impending AI revolution and the rapidly evolving tech landscape. NLP consistently features as the fastest-growing skill in demand by Upwork (largest freelancing platform), and the job listings with an NLP tag continue to feature prominently on various job boards. Since NLP is a subfield of ML, organizations typically hire candidates as ML engineers to work on NLP projects. You could be working on the most cutting-edge ideas in large technology firms or implementing NLP technology-based applications in banks, e-commerce organizations, and so on. The exact work performed by NLP engineers can vary from project to project. However, working with large volumes of unstructured data, preprocessing data, reading research papers on the new development in the field, tuning model parameters, continuous improvement, and so on are some of the tasks that are commonly performed. The authors, having worked on several NLP projects and having followed the latest industry trends closely, can safely state that it's a very exciting time to work in the field of NLP. You can benefit from learning about NLP even if you are simply a tech enthusiast and not particularly looking for a job as an NLP engineer. You can expect to build reasonably sophisticated NLP applications and tools on your MacBook or PC, on a shoestring budget. It is

not surprising, therefore, that there has been a surge of start-ups providing NLP-based solutions to enterprises and retail clients.

A few of the exciting start-ups in this area are listed as follows:

- **Luminance:** Legal tech start-up aimed at analyzing legal documents
- **NetBase:** Real-time social media feed analytics
- **Agolo:** Summarizes large bodies of text at scale.
- **Idibon:** Converts unstructured data to structured data.

This area is also witnessing brisk acquisition activities with larger tech companies acquiring start-ups (Samsung acquired Kngine; Reliance Communications acquired chatbot start-up Haptik; and so on). Given the low barriers for entry and easily accessible open source technologies, this trend is expected to continue. Now that we have familiarized ourselves with NLP and the benefits of gaining proficiency in this area, we will discuss the current and evolving applications of NLP.

1.2 Current applications of NLP

NLP applications are everywhere, and it is highly unlikely that you have not interacted with any such application over the past few days. The current applications include virtual assistants (Alexa, Siri, Cortana, and so on), customer support tools (chatbots, email routers/classifiers, and so on), sentiment analyzers, translators, and document ranking systems. The adoption of these tools is quickly growing, since the speed and accuracy of these applications have increased manifold over the years. It should be noted that many popular NLP applications such as, Alexa and conversational bots, need to process audio data, which can be quantified by capturing the frequency of the underlying sound waves of the audio. For these applications, the data preprocessing steps are different from those for a text-based application, but the core principles of analyzing the data remain the same and will be discussed in detail in this book. The following are examples of some widely used NLP tools. These tools could be web applications or desktop applications with which you can interact via the user interface. We will be covering the models powering these tools in detail in the subsequent chapters.

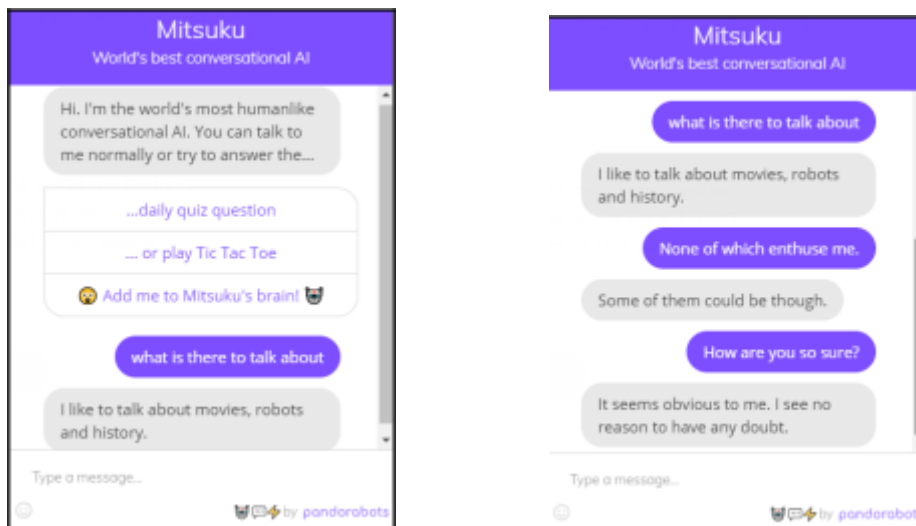
1.3 Chatbots

Chatbots are AI-based software that can conduct conversations with humans in natural languages. Chatbots are used extensively as the first point of customer support and have been very effective in resolving simple user queries. As per industry estimates, the size of the global chatbot market is expected to grow to \$102 billion by 2025, compared to the market size of \$17 billion in 2019 (source:

<https://www.mordorintelligence.com/industry-reports/chatbot-market>). The significant savings generated by these chatbots for organizations is the major driver for the increase in the uptake of this technology.

Chatbots can be simple and rule-based, or highly sophisticated, depending on business requirements. Most chatbots deployed in the industry today are trained to direct users to the appropriate source of information or respond to queries pertaining to a specific subject. It is highly unlikely to have a generalist chatbot capable of fielding questions pertaining to a number of areas. This is because training a chatbot on a given topic requires a copious amount of data, and training on a number of topics could result in performance issues.

The next screenshots are from my conversation with one of the smartest chatbots available, named Mitsuku (<https://www.pandorabots.com/mitsuku/>). The Mitsuku chatbot was created by Steve Worswick and it has the distinction of winning the Loebner Prize multiple times due to it being adjudged the most human-like AI application. The application was created using **Artificial Intelligence Markup Language (AIML)** and is mostly a rule-based application. Have a look at the following screenshots:



As you can see, this bot is able to hold simple conversations, just like a human. However, once you start asking technical questions or delve deeper into a topic, the quality of the responses deteriorates. This is expected, though, and we are still some time away from full human-like chatbots. You are encouraged to try engaging with Mitsuku in both simple and technical conversations and judge the accuracy yourself.

1.4 Sentiment Analysis

Sentiment analysis is a set of algorithms and techniques used to detect the sentiment (positive, negative, or

neutral) of a given text. This is a very powerful application of NLP and finds usage in a number of industries. Sentiment analysis has allowed entities to mine opinions from a much wider audience at significantly reduced costs. The traditional way of garnering feedback for companies has been through surveys, closed user group testing, and so on, which could be quite expensive. However, organizations can reduce costs by scraping data (from social media platforms or review-gathering sites) and using sentiment analysis to come up with an overall sentiment index of their products.

Here are some other examples of use cases of sentiment analysis:

- A stock investor scanning news about a company to assess overall market sentiment.
- An individual scanning tweets about the launch of a new phone to decide the prevailing sentiment.
- A political party analyzing social media feeds to assess the sentiment regarding their candidate.

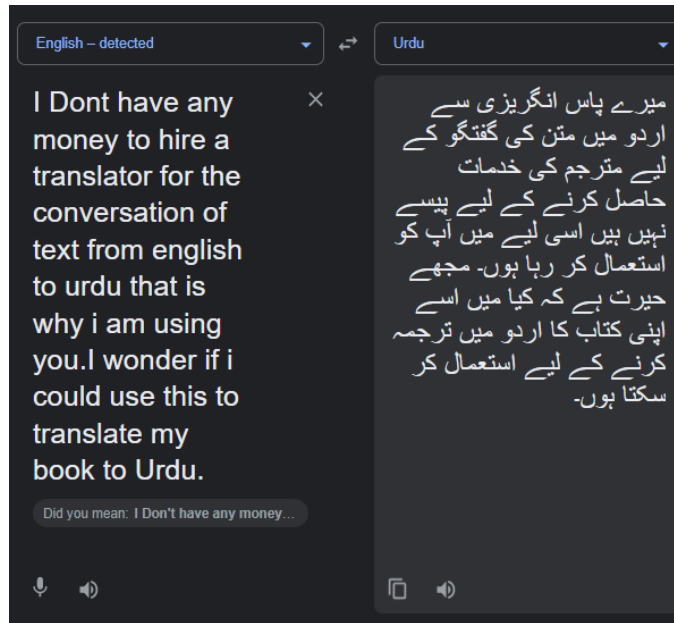
Sentiment analyzing systems can be simple lexicon-based or ML/DL-based. The choice of method depends on business requirements, the pros and cons of each approach, and other development constraints. This book will cover ML/DL-based methods in detail. A simple Google search yields numerous online sentiment analyzing sources, such as paralleldots.com. It's encouraged to try submitting sentences or paragraphs to the tool and analyze the response. These tools will likely do a reasonably good analysis of simple sentences or articles but output for sentences with complex structures (double negation, rhetorical questions, qualifiers, etc.) may not be accurate. Before using a pre-built sentiment analyzer, it is important to understand the methodology and training dataset used to build that analyzer. You don't want to use a sentiment analyzer trained on movie review data to predict the sentiment of text from a different area (such as financial news articles or restaurant reviews), as words that carry a positive or negative context in one area may have a neutral or opposite polarity context in another area. For example, some words signifying a positive sentiment in financial news articles are bullish, green, expansion, and growth, but these words, if used in a movie review context, would not be polarity-influencing words. Therefore, it is important to use suitable training data to build a sentiment analyzer.

We will delve deeper into sentiment analysis in Chapter 7, Identifying Patterns in Text Using Machine Learning, and will build a sentiment analyzer using product review data.

1.5 Machine translation:

Language translation was one of the early problems NLP techniques attempted to solve. During the Cold War, there was a pressing need to translate Russian documents into English using AI techniques. In 1964, the US government created the Automatic Language Processing Advisory Committee (ALPAC) of leading scientists, linguists, and researchers to explore the feasibility of machine translation, but they were unable to make any significant breakthrough, leading to skepticism around the feasibility of AI technology and a reduced interest in AI research in the 1970s, known as the AI Winter. Today, we have translators with high accuracy due to the DL and pattern detection approach in modern NLP. The high market value of the translation industry in the era of interconnected communities and global businesses is evident, and although businesses still rely mostly on human translators for important documents, the use of NLP techniques for conversation translation is increasing. Google

Translate uses an ANN-based system that predicts the possible sequence of translated words. A quick test of Google Translate's accuracy in translating text from English to Urdu is shown in a screenshot.



1.6 Important Python libraries

We will now discuss some of the most important Python libraries for NLP. We will delve deeper into some of these libraries in subsequent chapters.

1.6.1 NLTK

The Natural Language Toolkit library (NLTK) is one of the most popular Python libraries for natural language processing. It was developed by Steven Bird and Edward Loper of the University of Pennsylvania. Developed by academics and researchers, this library is intended to support research in NLP and comes with a suite of pedagogical resources that provide us with an excellent way to learn NLP. We will be using NLTK throughout this book, but first, let's explore some of the features of NLTK. However, before we do anything, we need to install the library by running the following command in the Anaconda Prompt:

- `pip install nltk`

1.6.2 NLTK corpora

A corpus is a large body of text or linguistic data and is very important in NLP research for application development and testing. NLTK allows users to access over 50 corpora and lexical resources (many of them mapped to ML-based applications). We can import any of the available corpora into our program and use NLTK functions to analyze the text in the imported corpus. More details about each corpus could be found here: <http://www.nltk.org/book/ch02.html>.

1.6.3 Text processing

As discussed previously, a key part of NLP is transforming text into mathematical objects. NLTK provides various functions that help us transform the text into vectors. The most basic NLTK function for this purpose is tokenization, which splits a document into a list of units. These units could be words, alphabets, or sentences. Refer to the following code snippet to perform tokenization using the NLTK library:

```
In [11]: from nltk.tokenize import sent_tokenize

In [12]: exam_sent = 'Hello AI 6th semester, This is sentence Tokenizing'

In [13]: exam_sent = 'Hello AI 6th semester, This is sentence Tokenizing'
          print(sent_tokenize(exam_sent))

['Hello AI 6th semester, This is sentence Tokenizing']

In [17]: from nltk.tokenize import word_tokenize

In [18]: exam_word = 'Hello AI 6th semester! This is word Tokenizing, I am really happy to do this!'
          print(word_tokenize(exam_word))

['Hello', 'AI', '6th', 'semester', '!', 'This', 'is', 'word', 'Tokenizing', ',', 'I', 'am', 'really', 'happy', 'to', 'do', 'this', '!', '']
```

We have tokenized the preceding sentence using the `word_tokenize()` function of NLTK, which is simply splitting the sentence by white space. The output is a list, which is the first step toward vectorization.

In our earlier discussion, we touched upon the computationally intensive nature of the vectorization approach due to the sheer size of the vectors. More words in a vector mean more dimensions that we need to work with. Therefore, we should strive to rationalize our vectors, and we can do that using some of the other useful NLTK functions such as stopwords, lemmatization, and stemming.

The following is a partial list of English stop words in NLTK. Stop words are mostly connector words that do not contribute much to the meaning of the sentence:

```
In [20]: stopwords=nlk.corpus.stopwords.words('english')
print(stopwords)

['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', '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', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'ar en', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "have n't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "should n't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

Since NLTK provides us with a list of stop words, we can simply look up this list and filter out stop words from our word list:

```
In [24]: newtokens=[word for word in tokens if word not in stopwords]
print (newtokens)

['Who', 'would', 'thought', 'computer', 'programs', 'would', 'analyzing', 'human', 'sentiments']
```

We can further modify our vector by using lemmatization and stemming, which are techniques that are used to reduce words to their root form. The rationale behind this step is that the imaginary n-dimensional space that we are navigating doesn't need to have separate axes for a word and that word's inflected form (for example, eat and eating don't need to be two separate axes). Therefore, we should reduce each word's inflected form to its root form. However, this approach has its critics because, in many cases, inflected word forms give a different meaning than the root word. For example, the sentences My manager promised me promotion and He is a promising prospect use the inflected form of the root word promise but in entirely different contexts. Therefore, you must perform stemming and lemmatization after considering its pros and cons.

The following code snippet shows an example of performing lemmatization using the NLTK library's WordNetlemmatizer module:

```
In [26]: text = "Who would have thought that computer programs would be analyzing human sentiments"
tokens = word_tokenize(text)
lemmatizer = WordNetLemmatizer()
tokens=[lemmatizer.lemmatize(word) for word in tokens]
print(tokens)

['Who', 'would', 'have', 'thought', 'that', 'computer', 'program', 'would', 'be', 'analyzing', 'human', 'sentiment']
```

Lemmatization is performed by looking up a word in WordNet's inbuilt root word map. If the word is not found, it returns the input word unchanged. However, we can see that the performance of the lemmatizer was not good and it was only able to reduce programs and sentiments from their plural forms. This shows that the lemmatizer is highly dependent on the root word mapping and is highly susceptible to incorrect root word transformation.

Stemming is similar to lemmatization but instead of looking up root words in a pre-built dictionary, it defines some rules based on which words are reduced to their root form. For example, it has a rule that states that any word with ing as a suffix will be reduced by removing the suffix.

The following code snippet shows an example of performing stemming using the NLTK library's PorterStemmer module:

```
In [27]: from nltk.stem import PorterStemmer
text = "Who would have thought that computer programs would be analyzing human sentiments"
tokens=word_tokenize(text.lower())
ps = PorterStemmer()
tokens=[ps.stem(word) for word in tokens]
print(tokens)

['who', 'would', 'have', 'thought', 'that', 'comput', 'program', 'would', 'be', 'analyz', 'human', 'sentiment']
```

As per the preceding output, stemming was able to transform more words than lemmatizing, but even this is far from perfect. In addition, you will notice that some stemmed words are not even English words. For example, *analyz* was derived from *analyzing* as it blindly applied the rule of removing *ing*.

The preceding examples show the challenges of reducing words correctly to their respective root forms using NLTK tools. Nevertheless, these techniques are quite popular for text preprocessing and vectorization. You can also create more sophisticated solutions by building on these basic functions to create your own lemmatizer and stemmer. In addition to these tools, NLTK has other features that are used for preprocessing, all of which we will discuss in subsequent chapters.

1.6.4 Part of speech tagging

Part of speech tagging (POS tagging) identifies the part of speech (noun, verb, adverb, and so on) of each word in a sentence. It is a crucial step for many NLP applications since, by identifying the POS of a word, we can deduce its contextual meaning. For example, the meaning of the word **ground** is different when it is used as a noun; for example, The ground was sodden due to rain, compared to when it is used as an adjective, for example, The restaurant's ground meat recipe is quite popular. We will get into the details of POS tagging and its applications, such as **Named Entity Recognizer (NER)**, in subsequent chapters.

Refer to the following code snippets to perform POS tagging using NLTK:

```
In [30]: nltk.pos_tag(["your"])
Out[30]: [('your', 'PRP$')]
```

```
In [28]: nltk.pos_tag(["beautiful"])
Out[28]: [('beautiful', 'NN')]
```

```
In [29]: nltk.pos_tag(["eat"])
Out[29]: [('eat', 'NN')]
```

We can pass a word as a list to the `pos_tag()` function, which outputs the word and its part of speech. We can generate POS for each word of a sentence by iterating over the token list and applying the `pos_tag()` function individually. The following code is an example of how POS tagging can be done iteratively:


```
In [31]: from nltk.tokenize import word_tokenize
text = "Usain Bolt is the fastest runner in the world"
tokens = word_tokenize(text)
[nltk.pos_tag([word]) for word in tokens]
```

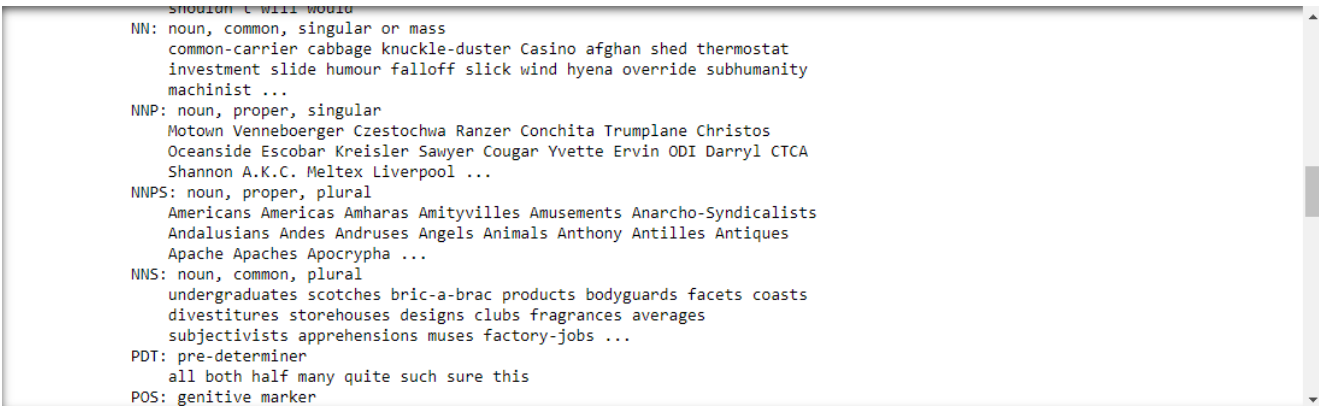
Here's the output:

```
Out[31]: [('Usain', 'NN')],
          [('Bolt', 'NN')],
          [('is', 'VBZ')],
          [('the', 'DT')],
          [('fastest', 'JJS')],
          [('runner', 'NN')],
          [('in', 'IN')],
          [('the', 'DT')],
          [('world', 'NN')]]
```

The exhaustive list of NLTK POS tags can be accessed using the `upenn_tagset()` function of NLTK:

```
In [32]: nltk.download('tagsets') # need to download first time
nltk.help.upenn_tagset()
```

Here is a partial screenshot of the output:



```

NN: noun, common, singular or mass
common-carrier cabbage knuckle-duster Casino afghan shed thermostat
investment slide humour falloff slick wind hyena override subhumanity
machinist ...
NNP: noun, proper, singular
Motown Venneboerger Czeszochwa Ranzer Conchita Trumplane Christos
Oceanside Escobar Kreisler Sawyer Cougar Yvette Ervin ODI Darryl CTCA
Shannon A.K.C. Meltex Liverpool ...
NNPS: noun, proper, plural
Americans Americas Amharas Amityvilles Amusements Anarcho-Syndicalists
Andalusians Andes Andruses Angels Animals Anthony Antilles Antiques
Apache Apaches Apocrypha ...
NNS: noun, common, plural
undergraduates scotches bric-a-brac products bodyguards facets coasts
divestitures storehouses designs clubs fragrances averages
subjectivists apprehensions muses factory-jobs ...
PDT: pre-determiner
all both half many quite sure this
POS: genitive marker
```

1.6.5 Textblob

Textblob is a popular library used for sentiment analysis, part of speech tagging, translation, and so on. It is built on top of other libraries, including NLTK, and provides a very easy-to-use interface, making it a must-have for NLP beginners. In this section, we would like you to dip your toes into this very easy-to-use, yet very versatile library. You can refer to Textblob's documentation, <https://textblob.readthedocs.io/en/dev/>, or visit its GitHub page, <https://github.com/sloria/TextBlob>, to get started with this library.

1.6.6 Part of speech tagging:

Textblob's POS tagging functionality is built on top of NLTK's tagging function, but with some modifications. You can refer to NLTK's documentation on POS tagging for more details: <https://www.nltk.org/book/ch05.html>. The tags function performs POS tagging like so:

```
In [5]: from textblob import TextBlob
TextBlob("The global economy is expected to grow this year").tags
```

Here is the output:

```
Out[5]: [('The', 'DT'),
         ('global', 'JJ'),
         ('economy', 'NN'),
         ('is', 'VBZ'),
         ('expected', 'VBN'),
         ('to', 'TO'),
         ('grow', 'VB'),
         ('this', 'DT'),
         ('year', 'NN')]
```

Since Textblob uses NLTK for POS tagging, the POS tags are the same as NLTK. This list can be accessed using the `upenn_tagset()` function of NLTK:

```
In [7]: import nltk
```

```
In [8]: nltk.download('tagsets') # need to download first time
nltk.help.upenn_tagset()
```

And the output will be this:

```
speculated wore appreciated contemplated ...
VBG: verb, present participle or gerund
telegraphing stirring focusing angering judging stalling lactating
hankerin' alleging veering capping approaching traveling besieging
encrypting interrupting erasing wincing ...
VBN: verb, past participle
multihulled dilapidated aerosolized chaired languished panelized used
experimented flourished imitated reunified factored condensed sheared
unsettled primed dubbed desired ...
VBP: verb, present tense, not 3rd person singular
predominate wrap resort sue twist spill cure lengthen brush terminate
appear tend stray glisten obtain comprise detest tease attract
emphasize mold postpone sever return wag ...
VBZ: verb, present tense, 3rd person singular
bases reconstructs marks mixes displeases seals carps weaves snatches
slumps stretches authorizes smolders pictures emerges stockpiles
seduces fizzes uses bolsters slaps speaks pleads ...
WDT: WH-determiner
that what whatever which whichever
```

These are just a few popular applications of Textblob and they demonstrate the ease of use and versatility of the program. There are many other applications of Textblob, and you are encouraged to explore them. A good place

to start your Textblob journey and familiarize yourself with other Textblob applications would be the Textblob tutorial, which can be accessed at <https://textblob.readthedocs.io/en/dev/quickstart.html>.

1.6.7 Tokenization with nltk:

In order to build up a vocabulary, the first thing to do is to break the documents or sentences into chunks called tokens. Each token carries a semantic meaning associated with it. Tokenization is one of the fundamental things to do in any text-processing activity. Tokenization can be thought of as a segmentation technique wherein you are trying to break down larger pieces of text chunks into smaller meaningful ones. Tokens generally comprise words and numbers, but they can be extended to include punctuation marks, symbols, and, at times, understandable emoticons. Let's go through a few examples to understand this better:

```
In [18]: exam_word = 'Hello AI 6th semester! This is word Tokenizing, I am really happy to do this!'
          print(word_tokenize(exam_word))

['Hello', 'AI', '6th', 'semester', '!', 'This', 'is', 'word', 'Tokenizing', ',', 'I', 'am', 'really', 'happy', 'to', 'do', 'this', '!']

In [19]: exam_word.split(' ')
```

Here's the output.

```
Out[19]: ['Hello',
          'AI',
          '6th',
          'semester!',
          'This',
          'is',
          'word',
          'Tokenizing,',
          'I',
          'am',
          'really',
          'happy',
          'to',
          'do',
          'this!']
```

A simple sentence.split() method could provide us with all the different tokens in the sentence The capital of China is Beijing. Each token in the preceding split carries an intrinsic meaning; however, it is not always as straightforward as this.

1.6.8 Issue with Tokenization

Consider the sentence and corresponding split in the following example:

```
In [9]: sentence = "China's capital is Beijing"
        sentence.split()
```

Here's the output:

```
Out[9]: ["China's", 'capital', 'is', 'Beijing']
```

In the preceding example, should it be China, Chinas, or China's? A split method does not often know how to deal with situations containing apostrophes. In the next two examples, how do we deal with we'll and I'm? We'll indicates we will and I'm indicates I am. What should be the tokenized form of we'll? Should it be well or we'll or we and 'll separately? Similarly, how do we tokenize I'm? An ideal tokenizer should be able to process we'll into two tokens, we and will, and I'm into two tokens, I and am. Let's see how our split method would do in this situation.

1.6.9 Stemming using nltk

Imagine bringing all of the words computer, computerization, and computerize into one word, compute. What happens here is called stemming. As part of stemming, a crude attempt is made to remove the inflectional forms of a word and bring them to a base form called the stem. The chopped-off pieces are referred to as affixes. In the preceding example, compute is the base form and the affixes are r, rization, and rize, respectively. One thing to keep in mind is that the stem need not be a valid word as we know it. For example, the word traditional would get stemmed to tradit, which is not a valid word in the English dictionary.

The two most common algorithms/methods employed for stemming include the Porter stemmer and the Snowball stemmer. The Porter stemmer supports the English language, whereas the Snowball stemmer, which is an improvement on the Porter stemmer, supports multiple languages, which can be seen in the following code snippet and its output:

```
In [14]: from nltk.stem.snowball import SnowballStemmer
```

```
In [15]: print(SnowballStemmer.languages)
('arabic', 'danish', 'dutch', 'english', 'finnish', 'french', 'german', 'hungarian', 'italian', 'norwegian', 'porter', 'portuguese', 'romanian', 'russian', 'spanish', 'swedish')
```

One thing to note from the snippet is that the Porter stemmer is one of the offerings provided by the Snowball stemmer. Other stemmers include the Lancaster, Dawson, Krovetz, and Lovins stemmers, among others. We will look at the Porter and Snowball stemmers in detail here.

The Porter stemmer works only with strings, whereas the Snowball stemmer works with both strings and Unicode data. The Snowball stemmer also allows the option to ignore stopwords as an inherent functionality.

Let's now first apply the Porter stemmer to words and see its effects in the following code block:

```
In [16]: plurals = ['caresses', 'flies', 'dies', 'mules', 'died', 'agreed', 'owned',  
                  'humbled', 'sized', 'meeting', 'stating',  
                  'siezing', 'itemization', 'traditional', 'reference', 'colonizer',  
                  'plotted', 'having', 'generously']
```

Here's the stemmed output from the Porter stemming algorithm:

```
In [17]: from nltk.stem.porter import PorterStemmer  
stemmer = PorterStemmer()  
singles = [stemmer.stem(plural) for plural in plurals]  
print(' '.join(singles))  
  
caress fli die mule die agre own humbl size meet state siez item tradit refer colon plot have gener
```

Next, let's see how the Snowball stemmer would do on the same text:

```
In [18]: stemmer2 = SnowballStemmer(language='english')  
singles = [stemmer2.stem(plural) for plural in plurals]  
print(' '.join(singles))  
  
caress fli die mule die agre own humbl size meet state siez item tradit refer colon plot have generous
```

As can be seen in the preceding code snippets, the Snowball stemmer requires the specification of a language parameter. In most of cases, its output is similar to that of the Porter stemmer, except for generously, where the Porter stemmer outputs gener and the Snowball stemmer outputs generous. The example shows how the Snowball stemmer makes minor changes to the Porter algorithm, achieving improvements in some cases.

1.6.10 Over-stemming and under-stemming

Potential problems with stemming arise in the form of over-stemming and understemming. A situation may arise when words that are stemmed to the same root should have been stemmed to different roots. This problem is referred to as over-stemming. In contrast, another problem occurs when words that should have been stemmed to the same root aren't stemmed to it. This situation is referred to as under-stemming. More about stemming can be read at <https://pdfs.semanticscholar.org/1c0c/0fa35d4ff8a2f925eb955e48d655494bd167.pdf>.

1.6.11 Lemmatization

Unlike stemming, wherein a few characters are removed from words using crude methods, lemmatization is a process wherein the context is used to convert a word to its meaningful base form. It helps in grouping together words that have a common base form and so can be identified as a single item. The base form is referred to as the lemma of the word and is also sometimes known as the dictionary form. Lemmatization algorithms try to identify the lemma form of a word by taking into account the neighborhood context of the word, part-of-speech (POS) tags, the meaning of a word, and so on. The neighborhood can span across words in the vicinity, sentences, or even documents. Also, the same words can have different lemmas depending on the context. A lemmatizer would try and identify the part-of-speech tags based on the context to identify the appropriate lemma. The most commonly used lemmatizer is the WordNet lemmatizer. Other lemmatizers include the Spacy lemmatizer, TextBlob lemmatizer, and Gensim lemmatizer, and others. In this section, we will explore the WordNet and Spacy lemmatizers.

1.6.12 WordNet lemmatizer

WordNet is a lexical database of English that is freely and publicly available. As part of WordNet, nouns, verbs, adjectives, and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing distinct concepts. These synsets are interlinked using lexical and conceptual semantic relationships. It can be easily downloaded, and the nltk library offers an interface to it that enables you to perform lemmatization. Let's try and lemmatize the following sentence using the WordNet lemmatizer:

Here is the code:

```
In [10]: import nltk
         nltk.download('wordnet')
         from nltk.stem import WordNetLemmatizer
         lemmatizer = WordNetLemmatizer()
         s = "We are putting in efforts to enhance our understanding of \
         Lemmatization"
         token_list = s.split()
         print("The tokens are: ", token_list)
         lemmatized_output = ' '.join([lemmatizer.lemmatize(token) for token \
         in token_list])
         print("The lemmatized output is: ", lemmatized_output)
```

And here is the output

```

lemmatizer = WordNetLemmatizer()
s = "We are putting in efforts to enhance our understanding of \
Lemmatization"
token_list = s.split()
print("The tokens are: ", token_list)
lemmatized_output = ' '.join([lemmatizer.lemmatize(token) for token \
in token_list])
print("The lemmatized output is: ", lemmatized_output)

The tokens are: ['We', 'are', 'putting', 'in', 'efforts', 'to', 'enhance', 'our', 'understanding', 'of', 'Lemmatization']
The lemmatized output is: We are putting in effort to enhance our understanding of Lemmatization

```

As can be seen, the WordNet lemmatizer did not do much here. Out of are, putting, efforts, and understanding, none were converted to their base form.

3.1.1.1 What are we lacking here?

The WordNet lemmatizer works well if the POS tags are also provided as inputs. It is really impossible to manually annotate each word with its POS tag in a text corpus. Now, how do we solve this problem and provide the part-of-speech tags for individual words as input to the WordNet lemmatizer? Fortunately, the nltk library provides a method for finding POS tags for a list of words using an averaged perceptron tagger, the details of which are out of the scope of this chapter. The POS tags for the sentence We are trying our best to understand Lemmatization here provided by the POS tagging method can be found in the following code snippet:

```

In [12]: nltk.download('averaged_perceptron_tagger')
pos_tags = nltk.pos_tag(token_list)
pos_tags

[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data] C:\Users\munee\AppData\Roaming\nltk_data...
[nltk_data] Package averaged_perceptron_tagger is already up-to-
[nltk_data] date!

Out[12]: [('We', 'PRP'),
          ('are', 'VBP'),
          ('putting', 'VBG'),
          ('in', 'IN'),
          ('efforts', 'NNS'),
          ('to', 'TO'),
          ('enhance', 'VB'),
          ('our', 'PRP$'),
          ('understanding', 'NN'),
          ('of', 'IN'),
          ('Lemmatization', 'NN')]

```

As can be seen, a list of tuples of the form (the token and POS tag) is returned by the POS tagger. Now, the POS tags need to be converted to a form that can be understood by the WordNet lemmatizer and sent in as input along with the tokens. The code snippet does what's needed by mapping the POS tags to the first character, which is accepted by the lemmatizer in the appropriate format:

```
In [23]: from nltk.corpus import wordnet
##This is a common method which is widely used across the NLP community of practitioners and readers
def get_part_of_speech_tags(token):
##Maps POS tags to first character Lemmatize() accepts. We are focusing on Verbs, Nouns, Adjectives and Adverbs here."""
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    tag = nltk.pos_tag([token])[0][1][0].upper()
    return tag_dict.get(tag, wordnet.NOUN)
```

Now, let's see how the WordNet lemmatizer performs when the POS tags are also provided as inputs:

```
In [24]: lemmatized_output_with_POS_information = [lemmatizer.lemmatize(token,
get_part_of_speech_tags(token)) for token in token_list]
print(' '.join(lemmatized_output_with_POS_information))
```

Here's the output:

```
We be put in effort to enhance our understand of Lemmatization
```

The following conversions happened:

- are to be
- putting to put
- efforts to effort
- understanding to understand

Let's compare this with the Snowball stemmer:

```
In [25]: stemmer2 = SnowballStemmer(language='english')
stemmed_sentence = [stemmer2.stem(token) for token in token_list]
print(' '.join(stemmed_sentence))
```

The following conversions happened:

```
we are put in effort to enhanc our understand of lemmat
```

As can be seen, the WordNet lemmatizer makes a sensible and context-aware conversion of the token into its base form, unlike the stemmer, which tries to chop the affixes from the token.

1.6.13 Spacy lemmatizer

The Spacy lemmatizer comes with pretrained models that can parse text and figure out the various properties of the text, such as POS tags, named-entity tags, and so on, with a simple function call. The prebuilt models identify the POS tags and assign a lemma to each token, unlike the WordNet lemmatizer, where the POS tags need to be explicitly provided. We can install Spacy and download the en model for the English language by running the following command from the command line:

pip install spacy && python -m spacy download en


```
(base) C:\Users\munee>pip install spacy && python -m spacy download en
Requirement already satisfied: spacy in c:\users\munee\anaconda3\lib\site-packages (3.2.3)
Requirement already satisfied: typer<0.5.0,>=0.3.0 in c:\users\munee\anaconda3\lib\site-packages (from spacy) (0.4.0)
Requirement already satisfied: srsly<3.0.0,>=2.4.1 in c:\users\munee\anaconda3\lib\site-packages (from spacy) (2.4.2)
Requirement already satisfied: setuptools in c:\users\munee\anaconda3\lib\site-packages (from spacy) (52.0.0.post20210125)
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in c:\users\munee\anaconda3\lib\site-packages (from spacy) (3.0.6)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in c:\users\munee\anaconda3\lib\site-packages (from spacy) (1.0.6)
Requirement already satisfied: wasabi<1.1.0,>=0.8.1 in c:\users\munee\anaconda3\lib\site-packages (from spacy) (0.9.0)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in c:\users\munee\anaconda3\lib\site-packages (from spacy) (2.0.6)
Requirement already satisfied: numpy>=1.15.0 in c:\users\munee\anaconda3\lib\site-packages (from spacy) (1.20.3)
Requirement already satisfied: packaging>=20.0 in c:\users\munee\anaconda3\lib\site-packages (from spacy) (21.0)
Requirement already satisfied: Jinja2 in c:\users\munee\anaconda3\lib\site-packages (from spacy) (2.11.3)
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in c:\users\munee\anaconda3\lib\site-packages (from spacy) (4.62.1)
```

Now that we have installed spacy, let's see how spacy helps with lemmatization using the following code snippet:

```
In [29]: import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("We are putting in efforts to enhance our understanding of Lemmatization")
" ".join([f"{token.lemma_}({token.pos_})" for token in doc])

Out[29]: 'we(PRON) be(AUX) put(VERB) in(ADP) effort(NOUN) to(PART) enhance(VERB) our(PRON) understanding(NOUN) of(ADP) Lemmatization(PRO
PN)'
```

The spacy lemmatizer performed a decent job without the input information of the POS tags. The advantage here is that there's no need to look out for external dependencies for fetching POS tags as the information is built into the pretrained model. Another thing to note in the preceding output is the -PRON- lemma. The lemma for Pronouns is returned as -PRON- in Spacy's default behavior. It can act as a feature or, conversely, can be a limitation since the exact lemma is not being returned. Spacy supports multiple languages other than English. You can learn what they are at <https://spacy.io/usage/models>.

1.6.14 Stopword removal

From time to time in the previous sections, a technique called stopwords removal was mentioned. We will finally look at the technique in detail here. What are stopwords? Stopwords are words such as a, an, the, in, at, and so on that occur frequently in text corpora and do not carry a lot of information in most contexts. These words, in general, are required for the completion of sentences and making them grammatically sound. They are often the most common words in a language and can be filtered out in most NLP tasks, and consequently help in reducing the vocabulary or search space. There is no single list of stopwords that is available universally, and they vary mostly based on use cases; however, a certain list of words is maintained for languages that can be treated as stopwords specific to that language, but they should be modified based on the problem that is being solved. Let's look at the stopwords available for English in the nltk library!

```
In [30]: nltk.download('stopwords')
from nltk.corpus import stopwords
stop = set(stopwords.words('english'))
", ".join(stop)

[nltk_data] Downloading package stopwords to
[nltk_data] C:\Users\munee\AppData\Roaming\nltk_data...
[nltk_data] Package stopwords is already up-to-date!
```

Here's the output:

```
Out[30]: "nor, yours, weren't, under, any, all, so, up, themselves, before, where, have, does, was, did, didn't, isn't, she's, an, the,
wasn't, most, below, this, shan't, hers, because, won't, you've, we, that'll, i, wouldn't, own, were, her, with, once, am, can,
doesn, more, won, itself, is, a, been, ain, aren, both, they, you'd, shouldn't, having, it's, are, against, our, further, were
n, theirs, until, ma, aren't, y, these, couldn't, haven, my, hadn, who, had, their, yourself, you're, mustn't, same, o, too, be
tween, shouldn, m, haven't, only, doesn't, them, from, whom, herself, d, me, there, myself, his, in, about, what, ourselves, a
s, him, hadn't, didn, through, over, will, such, should, ve, mightn, needn, yourselves, hasn, that, if, by, after, other, to, s
han, off, on, above, for, of, those, while, you, how, hasn't, re, should've, wasn, your, some, it, don't, down, couldn, wouldn,
not, but, or, be, during, mightn't, at, here, being, doing, just, its, each, mustn, and, now, again, ours, which, than, himsel
f, he, no, why, when, isn, needn't, do, ll, you'll, few, very, has, don, into, out, then, she, s, t"
```

If you look closely, you'll notice that Wh- words such as who, what, when, why, how, which, where, and whom are part of this list of stopwords; however, in one of the previous sections, it was mentioned that these words are very significant in use cases such as question answering and question classification. Measures should be taken to ensure that these words are not filtered out when the text corpus undergoes stopwords removal. Let's learn how this can be achieved by running through the following code block:

```
wh_words = ['who', 'what', 'when', 'why', 'how', 'which', 'where', 'whom']
stop = set(stopwords.words('english'))

sentence = "how are we putting in efforts to enhance our understanding of Lemmatization"
for word in wh_words:
    stop.remove(word)
sentence_after_stopword_removal = [token for token in sentence.split() if
token not in stop]
" ".join(sentence_after_stopword_removal)
```

Here's the output:

```
Out[34]: 'how putting efforts enhance understanding Lemmatization'
```

The preceding code snippet shows that the sentence how are we putting in efforts to enhance our understanding of Lemmatization gets modified to how putting efforts enhance understanding Lemmatization. The stopwords are, we, in, to, our, and of were removed from the sentence. Stopword removal is generally the first step that is taken after tokenization while building a vocabulary or preprocessing text data.