Introduction to NLP & Lexical Processing

Module 1 : Introduction to NLP

Introduction:-

Welcome to the module on Lexical Processing. Natural language processing, also referred to as text analytics, plays a very vital role in today's era because of the sheer volume of text data that users generate around the world on digital channels such as social media apps, e-commerce websites, blog posts, etc. Lexical Processing forms a basis for Natural language processing where you will convert the raw text into words and, depending on your application's needs, into sentences or paragraphs as well.

The first session of this module will take you through the following lectures:

- Industry applications of text analytics
- Understanding textual data
- Regular expressions

In the second session, you will learn how to clean the raw textual data and convert them into numeric features which can further be used to build machine learning models.

Please note that the terms 'natural language processing' and 'text analytics' will be used interchangeably in this module. However, they refer to the same entity.

NLP: Areas of Application:-

Before diving into what is textual data and how to handle it, let's take a look at the different industries that make use of text data to solve important problems.

So those were the different areas where text analytics is used extensively. In the next segment, you'll understand how the course on text analytics is structured.

Understanding Text:-

As you learnt in the previous section, NLP has a pretty wide array of applications - it finds use in many fields such as social media, banking, insurance and many more.

However, there is one question that still remains. The data you'll get while performing analytics on text, very often, will be just a sequence of words. Something like the text shown in the image below:

History [edit]

The history of natural language processing generally started in the 1950s, although work can be found from earlier periods. In 1950, Alan Turing published an article titled "Computing Machinery and Intelligence" which proposed what is now called the Turing test as a criterion of intelligence.

The Georgetown experiment in 1954 involved fully automatic translation of more than sixty Russian sentences into English. The authors claimed that within three or five years, machine translation would be a solved problem.[2] However, real progress was much slower, and after the ALPAC report in 1966, which found that ten-year-long research had failed to fulfill the expectations, funding for machine translation was dramatically reduced. Little further research in machine translation was conducted until the late 1980s, when the first statistical machine translation systems were developed.

Some notably successful natural language processing systems developed in the 1960s were SHRDLU, a natural language system working in restricted "blocks worlds" with restricted vocabularies, and ELIZA, a simulation of a Rogerian psychotherapist, written by Joseph Weizenbaum between 1964 and 1966. Using almost no information about human thought or emotion, ELIZA sometimes provided a startlingly human-like interaction. When the "patient" exceeded the very small knowledge base, ELIZA might provide a generic response, for example, responding to "My head hurts" with "Why do you say your head hurts?".

Now, think about it, if the data you get is of this form, and your task is to create an algorithm that translates this paragraph to a different language, say, Hindi, then how exactly will you do it?

To do so, your system should be able to take the raw unprocessed data shown above, break the analysis down into smaller sequential problems (a pipeline), and solve each of those problems individually. The individual problems could be as simple as breaking the data into sentences, words etc. to something as complex as understanding what a word means, based on the words in its “neighbourhood”.

Now, let’s listen to the professor as he talks about different levels of text analytics.

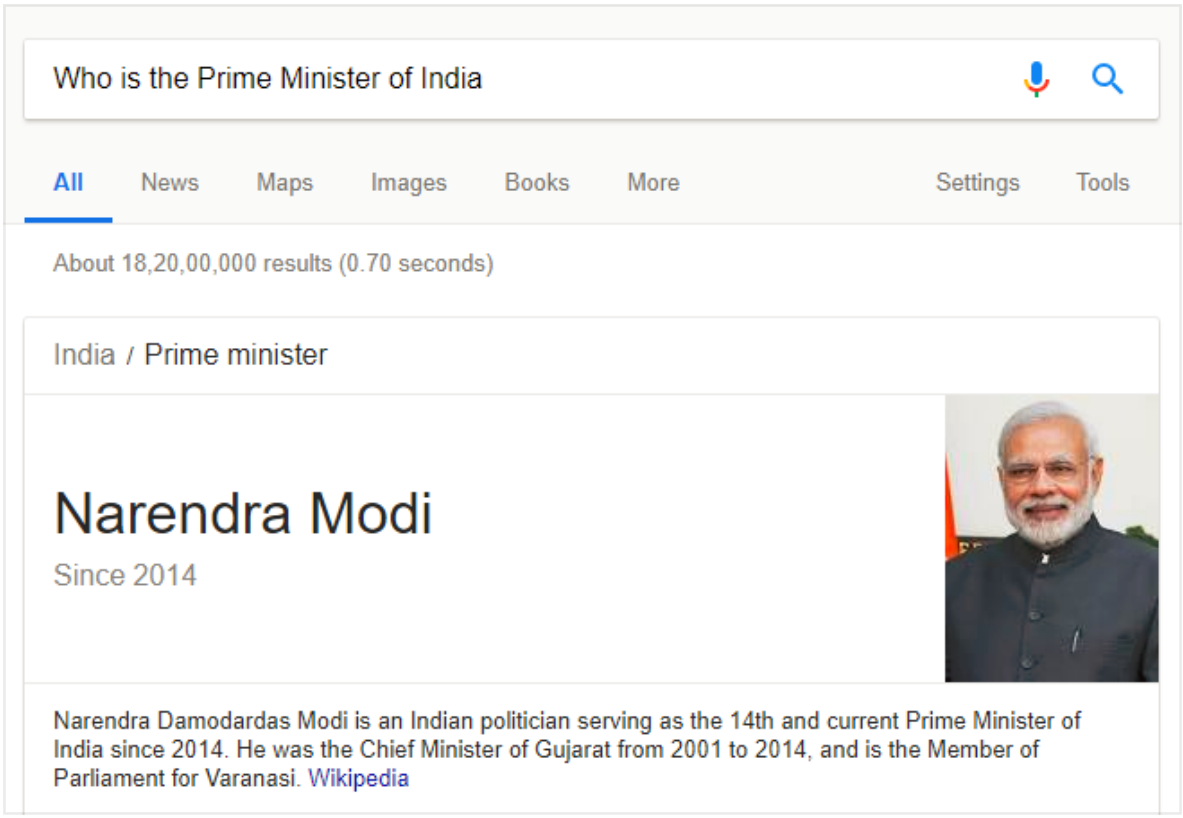
In the above video, you came across the different “steps” generally undertaken on the journey from data to meaning. Depending on the type of application you choose the extent to which the text processing is to be done. In this capstone, to build a text classification model, you will be predominantly using the lexical processing techniques and this module will help you acquire all the necessary lexical processing skills.

Now that you have looked at the areas of text analytics, let's take a brief look at what does it mean to understand the text, i.e., how to approach a problem that deals with text.

Let's go back to the Wikipedia example. Recall what the data (textual data) looked like - it was simply a collection of characters, that machines can't make any sense of. Starting with this data, you will move according to the following steps -

- **Lexical Processing:** First, you will just convert the raw text into words and, depending on your application's needs, into sentences or paragraphs as well.
    1. For example, if an email contains words such as lottery, prize and luck, then the email is represented by these words, and it is likely to be a spam email.
    2. Hence, in general, the group of words contained in a sentence gives us a pretty good idea of what that sentence means. Many more processing steps are usually undertaken in order to make this group more representative of the sentence; for example, cat and cats are considered to be the same word. In general, we can consider all plural words to be equivalent to the singular form.
    3. For a simple application like spam detection, sentiment classification lexical processing works just fine. Still, it is usually not enough in more complex applications, like, say, machine translation we need to go ahead with syntactic/semantic processing. For example, the sentences "My cat ate its third meal" and "My third cat ate its meal", have very different meanings. However, lexical processing will treat the two sentences as equal, as the "group of words" in both sentences is the same.
- **Syntactic Processing:** So, the next step after the lexical analysis is where we try to extract more meaning from the sentence, by using its syntax this time. Instead of only looking at the words, we look at the syntactic structures, i.e., the grammar of the language to understand what the meaning is.
    1. One example is differentiating between the subject and the object of the sentence, i.e., identifying who is performing the action and who is the person affected by it. For example, "Ram thanked Shyam" and "Shyam thanked Ram" are sentences with different meanings from each other because in the first instance, the action of 'thanking' is done by Ram and affects Shyam, whereas, in the other one, it is done by Shyam and affects Ram. Hence, a syntactic analysis that is based on a sentence's subjects and objects will be able to make this distinction.
    2. There are various other ways in which these syntactic analyses can help us enhance our understanding. For example, a question

answering system that is asked the question “Who is the Prime Minister of India?”, will perform much better, if it can understand that the words “Prime Minister” are related to “India”. It can then look up in its database, and provide the answer.



- **Semantic Processing**: Lexical and syntactic processing don't suffice when it comes to building advanced NLP applications such as language translation, chatbots etc... The machine, after the two steps given above, will still be incapable of actually understanding the meaning of the text. Such incapability can be a problem for, say, a question answering system, as it may be unable to understand that PM and Prime Minister mean the same thing. Hence, when somebody asks it the question, “Who is the PM of India?”, it may not even be able to answer unless it has a separate database for PMs, as it won’t understand that the words PM and Prime Minister are the same. You could store the answer separately for both the variants of the meaning (PM and Prime Minister), but how many of these meanings are you going to store manually? At some point, your machine should be able to identify synonyms, antonyms, etc. on its own.
    1. This is typically done by inferring the word’s meaning to the collection of words that usually occur around it. So, if the words, PM and Prime Minister occur very frequently around similar words, then you can assume that the meanings of the two words are similar as well.

2. In fact, this way, the machine should also be able to understand other semantic relations. For example, it should be able to understand that the words “King” and “Queen” are related to each other and that the word “Queen” is simply the female version of the word “King”. Also, both of these words can be clubbed under the word “Monarch”. You can probably save these relations manually, but it will help you a lot more, if you can train your machine to look for the relations on its own, and learn them. Exactly how that training can be done, is something we’ll explore in the third module.

Once you have the meaning of the words, obtained via semantic analysis, you can use it for a variety of applications. Machine translation, chatbots and many other applications require a complete understanding of the text, right from the lexical level to the understanding of syntax to that of meaning. Hence, in most of these applications, lexical and syntactic processing simply form the “pre-processing” layer of the overall process. In some simpler applications, only lexical processing is also enough as the pre-processing part.
This gives you a basic idea of the process of analysing text and understanding the meaning behind it. Now, in the next segment, you'll learn how text is stored on machines.

Text Encoding:-

Data is being collected in many languages. However, in this course, you will be doing text analysis for the English language. The text analytics techniques that work for English might not work for other languages.

Let’s have Prof. Srinath discuss this. He explains how characters of different languages are stored on computers.

Now, it is not necessary that when you work with text, you’ll get to work with the English language. With so many languages in the world and internet being accessed by many countries, there is a lot of text in non-English languages. For you to work with non-English text, you need to understand how all the other characters are stored.

Computers could handle numbers directly and store them on registers (the smallest unit of memory on a computer). But they couldn’t store the non-numeric characters as is. The alphabets and special characters were to be converted to a numeric value first before they could be stored.

Hence, the concept of **encoding** came into existence. All the non-numeric characters were encoded to a number using a code. Also, the encoding techniques had to be standardised so that different computer manufacturers won’t use different encoding techniques.

The first encoding standard that came into existence was the **ASCII (American Standard Code for Information Interchange) standard**, in 1960. ASCII standard assigned a unique code to each character of the keyboard which was known as **ASCII code**. For example, the ASCII code of the alphabet 'A' is 65 and that of the digit zero is 48. Since then, there have been several revisions made to the codes to incorporate new characters that came into existence after the initial encoding.

When ASCII was built, English alphabets were the only alphabets that were present on the keyboard. With time, new languages began to show up on keyboard sets which brought new characters. ASCII became outdated and couldn't incorporate so many languages. A new standard has come into existence in recent years - the **Unicode standard**. It supports all the languages in the world - both modern and the older ones.

For someone working on text processing, knowing how to handle encodings becomes crucial. Before even beginning with any text processing, you need to know what kind of encoding the text has and if required, modify it to another encoding format.

In this segment, you'll understand how encoding works in Python and the different types of encodings that you can use in Python.

To get a more in-depth understanding of Unicode, there's a guide on official Python website. You can check it out here.

To summarise, there are two most popular encoding standards:

1. American Standard Code for Information Interchange (ASCII)
2. Unicode
    - UTF-8
    - UTF-16

Let's look at the relation between ASCII, UTF-8 and UTF-16 through an example. The table below shows the ASCII, UTF-8 and UTF-16 codes for two symbols - the dollar sign and the Indian rupee symbol.

Symbol	ASCII code		UTF-8 code		UTF-16 (BE) code	
	Binary	Hex	Binary	Hex	Binary	Hex
$ (Dollar sign)	00100100	24	00100100	24	00000000 00100100	0024
₹ (Indian Rupee sign)	Doesn't exist	Doesn't exist	11100010 10000010 10111001	E282B9	00100000 10111001	20B9

As you can see, UTF-8 offers a big advantage in cases when the character is an English character or a character from the ASCII character set. Also, while UTF-8 uses only 8 bits to store the character, UTF-16 (BE) uses 16 bits to store it, which looks like a waste of memory.

However, in the second case, a symbol is used which doesn't appear in the ASCII character set. For this case, UTF-8 uses 24 bits, whereas UTF-16 (BE) only uses 16. Hence the storage advantages offered by UTF-8 is reversed and actually becomes a disadvantage here. Also, the advantage UTF-8 offered previously by being same as the ASCII code is also not of use here, as ASCII code doesn't even exist for this case.

The default encoding for strings in python is Unicode UTF-8. You can also look at this UTF-8 encoder-decoder to look how a string is stored. Note that, the online tool gives you the hexadecimal codes of a given string.

Try this code in your Jupyter notebook and look at its output. Feel free to tinker with the code.

```
# create a string
amount = u"₹50"
print('Default string: ', amount, '\n', 'Type of string', type(amount), '\n')

# encode to UTF-8 byte format
amount_encoded = amount.encode('utf-8')
print('Encoded to UTF-8: ', amount_encoded, '\n', 'Type of string',
type(amount_encoded), '\n')


# sometime later in another computer...
# decode from UTF-8 byte format
amount_decoded = amount_encoded.decode('utf-8')
print('Decoded from UTF-8: ', amount_decoded, '\n', 'Type of string',
type(amount_decoded), '\n')
```

In the next segment, you'll learn about **regular expressions** which are a must-know tool for anyone working in the field of natural language processing and

text analytics.

Regular expressions: Quantifiers - I:-

This section onwards, you'll learn about **regular expressions**. Regular expressions, also called **regex**, are very powerful programming tools that are used for a variety of purposes such as feature extraction from text, string replacement and other string manipulations. For someone to become a master at text analytics, being proficient with regular expressions is a must-have skill.

A regular expression is a set of characters, or a **pattern**, which is used to find substrings in a given string.

Let's say you want to extract all the hashtags from a tweet. A hashtag has a fixed pattern to it, i.e. a pound ('#') character followed by a string. Some example hashtags are - #mumbai, #bangalore, #upgrad. You could easily achieve this task by providing this pattern and the tweet that you want to extract the pattern from (in this case, the pattern is - any string starting with #). Another example is to extract all the phone numbers from a large piece of textual data.

In short, if there's a pattern in any string, you can easily extract, substitute and do all kinds of other string manipulation operations using regular expressions.

Learning regular expressions basically means learning how to identify and define these patterns.

Regulars expressions are a language in itself since they have their own compilers. Almost all popular programming languages support working with regexes and so does Python.

Let's take a look at how to work with regular expressions in Python. Download the Jupyter Notebook provided below to follow along:

General Note on Practice Coding Questions

In the practice questions that you'll attempt in this module, the phrases 'match string' and 'extract string' will be used interchangeably. In both cases, you need to use the 're.search()' function which detects whether the given regular expression pattern is present in the given input string. The 're.search()' method returns a RegexObject if the pattern is found in the string, else it returns a None object.

After writing your code, you can use the 'Verify' button to evaluate your code against sample test cases. After verifying the code, you can 'Submit' the code, which will be then validated against the (hidden) test cases.

The comments in the coding questions will guide you with these nuances. Also, you can look at the sample solution after submitting your code (i.e. after the maximum number of allowed submissions) at the bottom of the coding console window.

So that's how you import regular expressions library in python and use it. You saw how to use the re.search() function - it returns a regex object if the pattern is found in the string. Also, you saw two of its methods - match.start() and match.end() which return the index of the starting and ending position of the match found.

Apart from re.search(), there are other functions in the re library that are useful for other tasks. You'll look at the other functions later in this session.

Now, the first thing that you'll learn about regular expressions is the use of **quantifiers**. Quantifiers allow you to mention and have control over how many times you want the character(s) in your pattern to occur.

Let's take an example. Suppose you have some data which have the word 'awesome' in it. The list might look like - ['awesome', 'awesomeeee', 'awesomee']. You decide to extract only those elements which have more than one 'e' at the end of the word 'awesome'. This is where quantifiers come into the picture. They let you handle these tasks.
You'll learn four types of quantifiers:

1. The '?' operator
2. The '*' operator
3. The '+' operator
4. The '{m, n}' operator

The first quantifier is '?'. Let's understand what the '?' quantifier does.

You heard Krishna say that you'll learn about five quantifiers instead of four. That's because the fourth quantifier has some more variations. You'll learn about it later in the session.

The '?' can be used where you want the preceding character of your pattern to be an **optional character in the string**. For example, if you want to write a regex that matches both 'car' and 'cars', the corresponding regex will be 'cars?'. 'S' followed by '?' means that 's' can be absent or present, i.e. it can be present zero or one time.

The next quantifier that you're going to study is the '*' quantifier.

A '*' quantifier matches the preceding character **any number of times**.
Practice some questions below to strengthen your understanding of it.

You learnt two quantifiers - the '?' and the '*'. In the next section, you'll learn

two more quantifiers.

Module 2 : Basic Lexical Processing

Module 3 : Advanced Lexical Processing