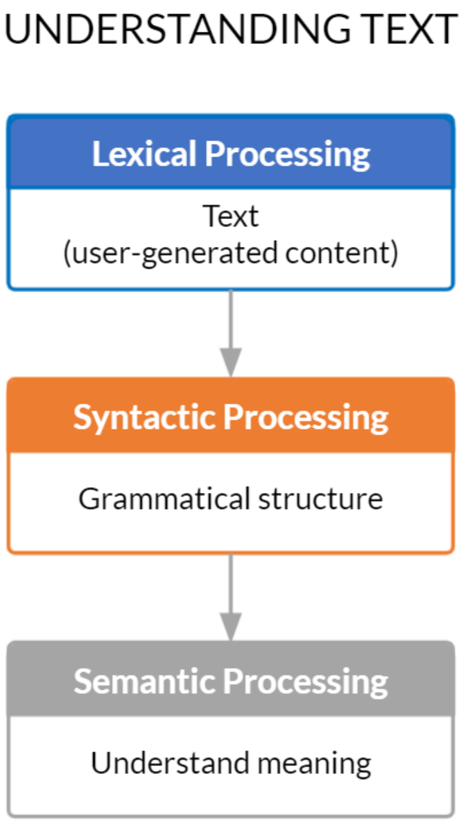
Summary

Fundamentals of NLP

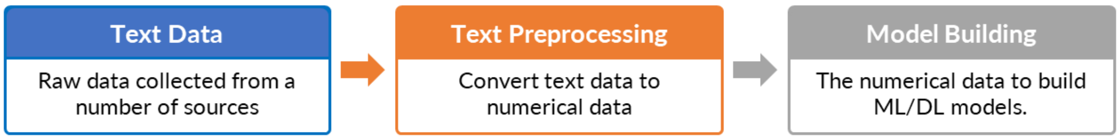
Overview of NLP

Natural language processing (NLP) is a sub-field of machine learning (ML) that deals with processing of linguistic data. NLP is a way to make a text machine understandable.

Language processing tools comprise three components that help convert text into machine understandable form.



All the NLP tools are data pre-processing tools that are meant to convert text into machine understandable form. The parts of NLP discussed earlier will not be able to build end-to-end ML solutions. The following flowchart shows the role of NLP in an ML solution.


Text Data
Raw data collected from a number of sources
Text Preprocessing
Convert text data to numerical data
Model Building
The numerical data to build ML/DL models.

Think of any example that you encounter in your daily life. For instance, consider the email classification that happens in Gmail. All your incoming mails are classified as primary social, promotional and so on. This is a classification task for which the solution must be built on some classification algorithm. The classification part is not special. The part that comes before classification, i.e., converting text data into a form which these algorithms can process, is the part where NLP tools come into picture.

Parts of NLP

Lexical Processing

The smallest segment of any text can be a word, a sentence or even a paragraph. This smallest unit is called a 'token'. In subsequent sections, you will learn the factors that drive the choice of the token. Lexical processing focusses on information extraction at the token level.

Lexical processing tools cannot differentiate between sentences that have similar words but different meanings. For instance, both the following sentences will have the same tokens, and hence, both the sentences are the same in terms of lexical processing, though they have different meanings.

Sentence A: "My cat ate its third meal".
Sentence B: "My third cat ate its meal".

Syntactic Processing

Syntactic processing is where you try to extract more information from the sentence by using its syntax and grammar. Instead of only looking at the words, you look at the grammatical structure of the sentence.

For instance, consider the word 'drive'. The word itself has a specific meaning, that is, 'to operate a motor vehicle'. However, if you consider a sentence like 'All the documents can be found in Google Drive', the word 'Drive' is being used as a noun.

Syntactic processing can be used to identify heteronyms, differentiate subject and object in a sentence and find name entities. There can be multiple other situations where the use of grammatical structure can lead to extraction of information that cannot be otherwise extracted using tokens.

Semantic Processing

Lexical and syntactic processing do not suffice when it comes to building advanced NLP applications, such as language translation and chatbots. Even after extracting information from the tokens and the grammar syntax, some information may still be lost. For example, if you ask a question answering system, “Who is the PM of India?”, it might not understand that the PM and the Prime Minister refer to the same entity. It might not even be able to give an answer unless it has a database connecting the PM to the Prime Minister. You could store the answer separately for both the variants of the meaning (PM and Prime Minister). However, how many such meanings will you store manually? At some point, your machine should be able to identify synonyms, antonyms, etc., on its own.

Applications of NLP

E-commerce

E-commerce businesses usually collect text information in the form of reviews from users. They can be product reviews, service reviews and so on. A lot of information can be extracted from these reviews. You can find the sentiment of the reviews, the major topics being spoken about, etc.

Healthcare

The healthcare industry is the most difficult to implement any of the ML solutions. The data is sensitive in nature. Most of it is in a non-digital form and labelling data from healthcare is extremely expensive. Having said that, healthcare can be a very important field for ML products. Healthcare has many NLP applications, such as treatment prediction and name entity extraction.

For instance, consider entity extraction. A language model needs to be able to identify a medical condition and the corresponding treatment from a piece of text. Once these models are trained enough, they can start recommending treatments to the doctor. Such recommendation systems can increase the capacity of doctors while examining patients.

Fintech

Fintech is always at the forefront of adopting new technologies such as NLP. One such example of NLP from the fintech world is CLEO. The application can go through a person's bank records and tag each of the expenditures into relevant categories, such as utilities, food and beverage and medicines. How do you think these tags are identified? Yes, a language model is trained to process the bank record entries and classify them into different groups. These classes can then be used to generate expenditure reports and so on.

Recommendation Systems

All the major tech companies, such as Facebook, Apple, Amazon, Netflix and Google (FAANG), use recommendation systems in some way or form.

Sentiment analysis on tweets.

Nowadays, large brands are highly cognizant of the image they have in the public. Consider any brand and it will have an image in your mind. These brands use data from social media platforms like Twitter and Facebook to maintain their image. NLP is substantially used to score the overall sentiment of people about a company. This analytics gives them the opportunity to take corrective measures if their image is changing.

Up to this point, you have learnt the NLP applications where input text is converted into information. But what about applications like Alexa? Do they also use NLP? Obviously yes. However, they also use a different part of NLP called natural language generation (NLG).

Any complete conversation has two parts – receiving information and responding to the information. Same is the case with chatbot solutions. They are built out of a combination of NLP and NLG.

Regular Expressions – Quantifiers

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.

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

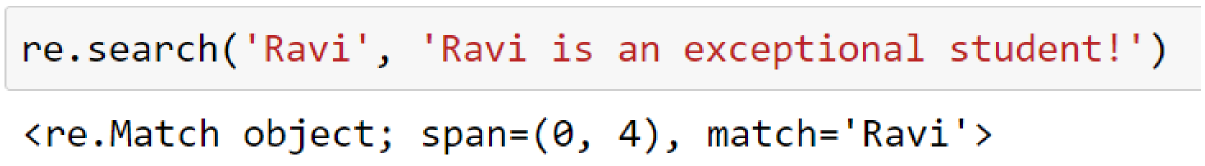
Suppose you want to extract all the hashtags from a tweet. A hashtag has a fixed pattern to it, i.e., a pound (‘#’) character followed by a string. For example, #India, #upgrad and #covid19. You could easily achieve this task by providing this pattern and the tweet that you want to extract the pattern from (in this case, the pattern is any string starting with #). Consider another example of extracting all the phone numbers from a large piece of textual data.

In short, if there is 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 means learning how to identify and define these patterns.

Regular expressions are a language in themselves since they have their own compilers. Almost all popular programming languages, including Python, support regexes.

Let’s take a look at an example of a simple pattern search.



```
re.search('Ravi', 'Ravi is an exceptional student!')
```

```
<re.Match object; span=(0, 4), match='Ravi'>
```

The ‘re.search()’ method is used to perform simple text searches. The first argument is the search key and the second argument is the corpus in which the search operation should be performed. Now, look at the printed result. It shows that a match has been found and provides the match position and the text that is matched.

Now, the first thing that you will learn regarding 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 a look at an example. Suppose you have some data that has the word ‘awesome’ in it. The list might look like [‘awesome’, ‘awesomeeee’, ‘awesomee’]. You decide to extract only those elements that have more than one ‘e’ at the end of the word ‘awesome’. This is where quantifiers come into picture. They let you handle such tasks.

The ‘?’ quantifier 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 either be absent or present one time.

The next quantifier is the ‘*’ quantifier. A ‘*’ quantifier causes the resulting regex to match 0 or more repetitions of the preceding regex.

For example, ‘ab*’ will positively match ‘a’, ‘ab’, ‘abbb’ and ‘a’ followed by any number of b’s. But, the same search ‘ab*’ will not match ‘aab’.

The ‘+’ quantifier causes the resulting regex to match one or more repetitions of the preceding regex. This means that the preceding character needs to be present **at least once** for the pattern to match the string.

For example, ‘ab+’ will successfully match ‘ab’ and ‘abb’, but it will not match ‘a’.

Thus, the only difference between ‘+’ and ‘*’ is that ‘+’ needs a character to be present **at least once** while ‘* does not.

To summarise, till now, you have learnt the following quantifiers:

- ‘?’: Optional preceding character
- ‘*’: Match preceding character repeating zero or more times
- ‘+’: Match preceding character repeating one or more times (i.e., at least once)

However, how do you specify a regex when you want to look for a character that appears, say, exactly five times or between three and five times? You cannot do that using the quantifiers you have learnt till now.

The next quantifier that you will learn will help you specify the number of occurrences of the preceding character.

Listed below are the four variants of this quantifier:

1. **{m, n}**: Matches the preceding character occurring ‘m’ to ‘n’ times
2. **{m, }**: Matches the preceding character occurring at least ‘m’ times, that is, there is no upper limit to the occurrence of the preceding character
3. **{, n}**: Matches the preceding character occurring zero to ‘n’ times, that is, the upper limit regarding the occurrence of the preceding character is fixed
4. **{n}**: Matches the preceding character occurring exactly ‘n’ number of times

Note that while specifying the {m,n} notation, avoid using a space after the comma, that is, use {m,n} rather than {m, n}.

An interesting thing to note is that this quantifier can replace the ‘?’, ‘*’ and ‘+’ quantifiers. That is because:

- '?' is equivalent to zero or once, that is, {0, 1},
- '*' is equivalent to zero or more times, that is, {0, }, and
- '+' is equivalent to one or more times, that is, {1, }.

Regular Expressions – Anchors and Wildcard

Anchors are used to specify the start and the end of the string. The ‘^’ symbol specifies the start of the string. The character followed by ‘^’ in the pattern should be the first character of the string in order for a string to match the pattern.

Similarly, the ‘$’ symbol specifies the end of the string. The character that precedes ‘$’ in the pattern should be the last character in the string for it to match the pattern.

Both the anchors can be specified in a single regex . For example, the regular expression pattern ‘^01*0$’ will match any string that starts and ends with zeroes with any number of 1s between them. It will match ‘010’, ‘0110’, ‘01111111110’ and even ‘00’ (‘*’ matches zero or more 1s). However, it will not match the string ‘0’ because there is only one 0 in this string, and in the pattern, you have specified that, there needs to be two 0s, one at the start and one at the end.

Now, there is one special character in regexes that acts as a placeholder and can match any character (literally) in the given input string. The ‘.’ (dot) character is also called the **wildcard character.**

Till now, you were mentioning the exact character followed by a quantifier in your regex patterns. For example, the pattern ‘hur{2,}ay’ matches ‘hurray’, ‘hurrray’, ‘hurrrray’ and so on. Here, the pattern specified that the letter ‘r’ should be present two or more times. However, you do not always know the letter that you want to repeat. In such situations, you use the wildcard instead.

The wildcard comes handy in many situations. It can be followed by a quantifier, which specifies that any character might be present a specified number of times.

For example, if you were to write a regex pattern that should match a string that starts with four characters followed by three 0s and two 1s and any two characters, the valid strings can be abcd00011ft, jkds00011hf, etc. The pattern that satisfies this kind of condition would be ‘.{4}0{3}1{2}.{2}’. You can also use ‘....00011..’ where the dot acts as a placeholder, which means that anything can come at the place of the dot. Both are correct regex patterns.

Until now, you were either using the actual letters (such as ab, 23 and 78) or the wildcard character in your regex patterns. There was no way of specifying whether the preceding character is a digit, an alphabet, a special character or a combination of these.

For example, suppose you want to match phone numbers in a large document. You know that the numbers may contain hyphens, plus symbols, etc. (e.g., +91-9930839123), but it will not have any letters of the alphabet. You will have to somehow specify that you are looking only for numerics and some other symbols, but not the letters of the alphabets.

To handle such situations, you can use **character sets**.

Character sets provide a lot more flexibility than just typing a wildcard or the literal characters. Character sets can be specified with or without a quantifier. When no quantifier follows the character set, it matches only one character and the match is successful only if the character in the string is one of the characters present inside the character set. For example, the pattern '[a-z]ed' will match strings such as 'ted', 'bed' and 'red' because the first character of each string, such as 't', 'b' and 'r', is present inside the range of the character set.

On the other hand, when you use a character set with a quantifier such as '[a-z]+ed', it will match any word that ends with 'ed' such as 'watched', 'baked', 'jammed' and 'educated'. In this way, a character set is similar to a wildcard because it can also be used with or without a quantifier. However, a character set gives you more power and flexibility.

Note that **a quantifier loses its special meaning** when it is present inside the character set. Inside square brackets, it is treated as any other character.

You can also mention a whitespace character inside a character set to specify one or more whitespaces inside the string. The pattern [A-z] can be used to match the full name of a person. It includes a space so it can match the full name which includes the person's first name, a space and the last name.

But what if you want to match every character other than the one mentioned inside the character set? You can use the caret operator for this purpose.

The '^' operator has two use cases. You already know that it can be used outside a character set to specify the start of a string. Here, it is known as an **anchor**.

Another use of this operator is inside a character set. When used inside a character set, it acts as a **complement operator**, i.e., it specifies that it will match any character other than those mentioned inside the character set.

The pattern [0-9] matches any single-digit number. On the other hand, the pattern '[^0-9]' matches any single character that is not a number.

Meta Sequences

When you work with regular expressions, you will find yourself using characters often. You will commonly use sets to match only digits, only alphabets, only alphanumeric characters, only whitespaces, etc.

Therefore, there is a shorthand way to write commonly used character sets in regular expressions. These are called meta-sequences.

You can use meta-sequences in the following two ways:

1. You can use them **without the square brackets**. For example, the pattern '\w+' will match any alphanumeric character.
2. You can also put them inside **the square brackets**. For example, the pattern '[\w]+' is the same as '\w+'. But when you use meta-sequences inside a square bracket, they are commonly used along with other meta-sequences. For example, the '[\w\s]+' matches both alphanumeric characters and whitespaces. The square brackets are used to group these two meta-sequences into one.

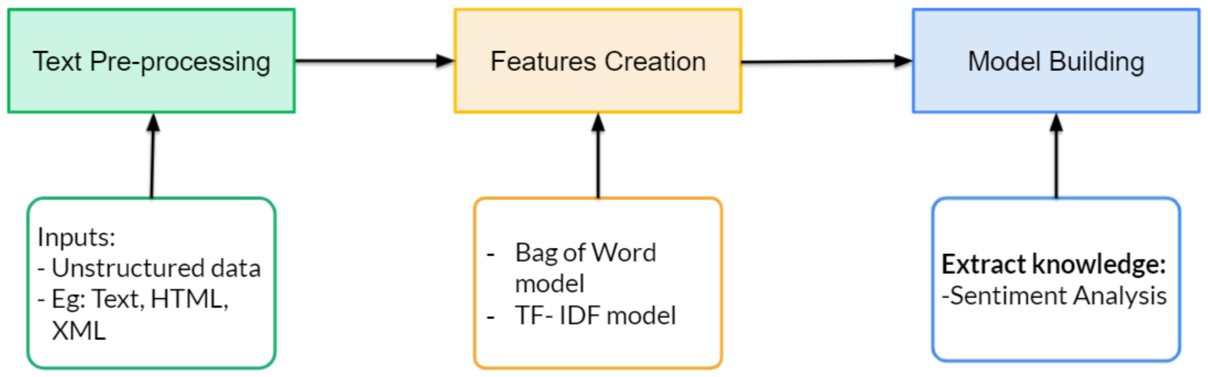
Pattern	Matches
[abc]	Matches either an a, b or c character
[abcABC]	Matches either an a, A, b, B, c or C character
[a-z]	Matches any characters between a and z, including a and z
[A-Z]	Matches any characters between A and Z, including A and Z
[a-zA-Z]	Matches any characters between a and z, including a and z ignoring cases of the characters
[0-9]	Matches any character which is a number between 0 and 9

upGrad

Summary

Basic Lexical Processing

The diagram given below shows the steps involved in creating an ML solution from text data.



User-generated text is usually extremely noisy – it has incorrect grammar, spelling mistakes, use of different cases and a lot of other inconsistencies. For instance, consider the following review written by a user for a phone that they bought from Amazon.

'Worst Phone I have seen specs amd buyed it but not satisfied I have samsung m21 mobile which has amoled screen and always on display same they mentioned in this Redmi Note 10s but display clarity is worst than LCD and always on display stays only for 10 sec .Worst Camera Performnace wise ok Didnt liked this phone and dont trap for technical gimmiks.'

Notice the spelling mistakes and grammatical errors in the text. The user also uses digits '21' to denote a specific phone model. All such inconsistencies need to be removed before a model can be built.

Text Encoding

The biggest challenge with text analytics is that machines cannot process human language; they can process only binary data, that is, 0s and 1s. Text encoding is a way to convert the letters of the alphabet, that is, human language (high-level) to machine language (binary).

The ASCII standard of text encoding was discussed. The ASCII (American Standard Code for Information Interchange) standard was the first encoding standard to come into existence. It assigns a unique code to all normal letters as well as digits and some special characters.

For example, the table provided below gives the ASCII code of a few characters.

Character	ASCII Code in Decimal	ASCII Code in Binary
A	65	01000001
p	112	01110000
%	37	00100101

Notice the binary form of ASCII characters. Each character is eight digits long, which means it takes eight bits, or one byte, to store each character in ASCII encoding. The standard ASCII character always begins with a 0, i.e., it is in the form of 0xxxxxxx. There are only 128 unique combinations of 0 and 1 in the form shown above.

When ASCII was built, only the letters of the English alphabet were used in the encoding. With time, as the need for more characters was felt, the encoding system had to be expanded.

The Unicode standard supports all the languages in the world, both modern and archaic.

To work on text processing, you need to know how to handle encoding. For instance, suppose you are encoding text about ‘Erwin Schrödinger’. If you use ASCII encoding, all the ö’s in the text will be lost. The encoder will not be able to convert this letter into machine language. So, before beginning with text processing, you need to know what kind of encoding the text has and, if required, modify it to another encoding format.

To get a more in-depth understanding regarding Unicode, you can refer to this guide available on the official Python website.

To summarise, the two most popular encoding standards include:

1. American Standard Code for Information Interchange (ASCII)
2. Unicode

Even Unicode is not the final version of the encoding standard. There are multiple versions of Unicode encoding, such as UTF-8, UTF-16 and UTF-32. Exploring these encoding techniques in detail is beyond the scope of this module.

UTF-8 offers a big advantage in cases when the character belongs to the English alphabet or the ASCII character set. Let’s take a look at the relation between ASCII and UTF-8 through an example. The table given below shows the encoding of two symbols on ASCII and UTF-8.

Character	Binary ASCII	Binary UTF-8
$	00100100	00100100
£	NA	11000010 10100011

For all characters present in ASCII encoding, the UTF code and the ASCII codes are the same. The characters that are not present in the ASCII set are encoded in two bytes. As the symbols become rarer, such as Greek letters and Chinese characters, the memory allocation of UTF-8 increases.

Dealing With Inconsistent Text

User-generated text, such as reviews, tweets, blogs and articles, have many inconsistencies, such as:

1. Converting text to a uniform case (lowercase)
2. Removing symbols and punctuations from text
3. Handling numbers in text

Case Standardisation

Why is it important to convert the complete text corpus to lower case?

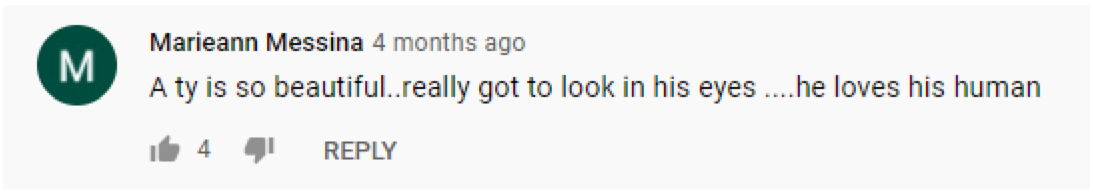
Note that lower case is not an absolute necessity; even if you convert the complete corpus to upper case, it will serve the same purpose. The objective of case conversion is to reduce variation in encoding. Earlier, you learnt that the binary versions of A and a are different regardless of the encoding technique being used. However, from a lingual perspective, A and a are not that different. So, it makes sense that they need to be encoded in similar binary forms. By converting the corpus into lower case, you can ensure that such variation is reduced.

Removal of Punctuations and Symbols

Next, Mahesh spoke about removing punctuation and symbols. Consider the following sentences:

1. I do not particularly like the play ‘Who’s Afraid of Virginia Woolf’?
2. I did not like it even when I worked at Yahoo!
3. I especially did not like it when I saw it at 5:00 am.

What kind of information is conveyed by '?' and the '!' in these sentences? They convey the tone of the sentence. In a grammatically correct text, such as a research paper or newspaper article, you may want to retain these characters because they are conveying certain information. On the other hand, take a look at the following comment by a user on YouTube.


Marieann Messina 4 months ago

A ty is so beautiful..really got to look in his eyes ....he loves his human

The comment is full of unnecessary full stops separating sentence fragments, not complete sentences. In such user-generated content, punctuation marks do not carry any meaning. Removing the punctuation marks and symbols makes this text cleaner.

Handling Numbers in a Text

Finally, let's discuss numbers embedded in text. In user-generated text, numbers perform different functions. They can be used to write words such as gr8 and 4ever. They can also denote quantity, such as 'I ate 2 pizzas' and 'There were 3 different curries on the plate'. And finally, numbers can be included in nouns, such as Brooklyn 99 and COVID-19.

You can either convert all numbers from digits to spellings or keep them as they are. The guiding principle to make this decision is: Redive the variation in encoding and preserve only the relevant information. For instance, consider the following sentence:

I bought 5 apples for ₹150, fruits have become quite expensive at this store.

If you need to build a model to guess the sentiment of this review, are the numbers necessary? The answer is 'no'. In this case, the negative sentiment is conveyed by words such as 'expensive'. On the other hand, consider the following text:

'Due to COVID-19, the 2020 Olympics were suspended.'

All the digits in this example are relevant, which is why you may want to keep them as they are.

Stop Word Removal

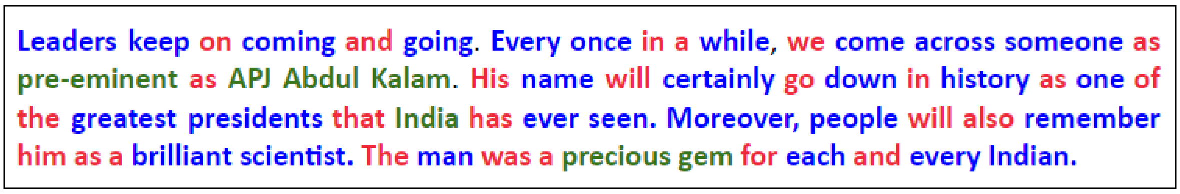
Consider any text corpus that is made of paragraphs, sentences and words. Some words are present for grammatical accuracy. Words such as 'is', 'an' and 'the' are mostly used to form meaningful sentences. Such words are called stop words.

Any text corpus contains three kinds of words:

- Highly frequent words called stop words, such as 'is', 'an' and 'the'

- Significant words, which are typically more important than the others to understand the text
- Rarely occurring words, which are less important than significant words

For example, consider the text given below. The words in red are stop words, those in blue are significant words and the ones in green are rare words.



Leaders keep on coming and going. Every once in a while, we come across someone as pre-eminent as APJ Abdul Kalam. His name will certainly go down in history as one of the greatest presidents that India has ever seen. Moreover, people will also remember him as a brilliant scientist. The man was a precious gem for each and every Indian.

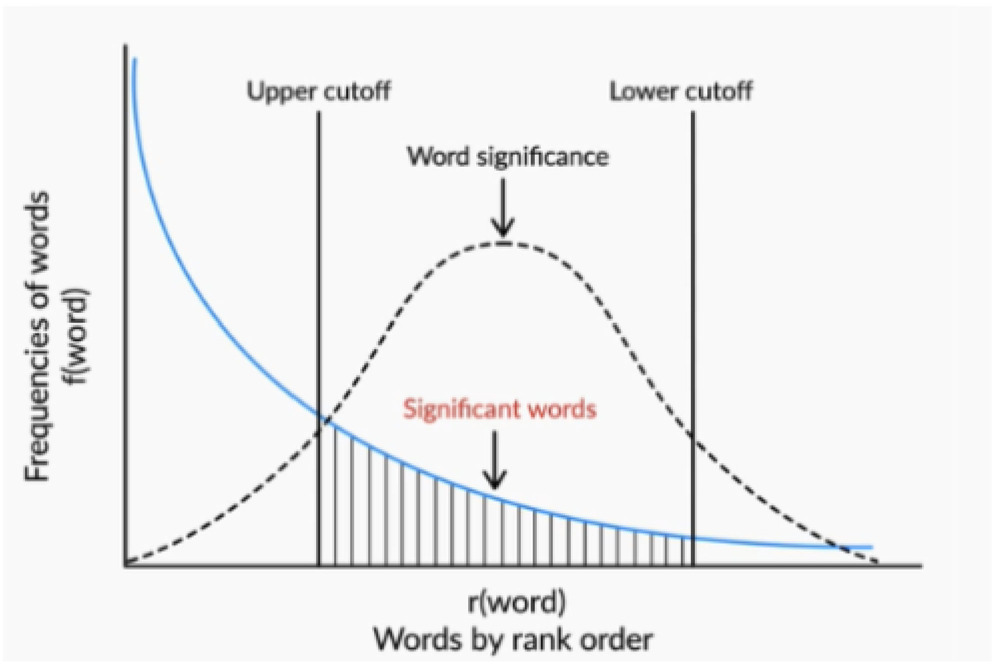
Generally speaking, stop words are removed from the text for the following two reasons:

1. They provide no useful information, especially in applications such as spam detectors or search engines. Therefore, you will remove stopwords from the spam data set.
2. Since the frequency of stop words is quite high, removing them significantly reduces the data size, which results in faster computation on text data and also reduces the number of features to deal with.

However, there are certain exceptions to these scenarios. In the next module, you will learn various concepts such as POS (parts-of-speech) tagging and parsing, where stop words are preserved because they provide meaningful (grammatical) information in some applications. Generally, stop words are removed unless they prove to be helpful in your application or analysis.

The most basic statistical analysis that you can do is to look at the **word frequency distribution**, i.e., visualise the word frequencies of a given text corpus. It turns out that you can see a common pattern when you plot word frequencies in a fairly large text corpus, such as a corpus of news articles, user reviews or Wikipedia articles.

Zipf's law (formulated by linguist-statistician George Zipf) states that the frequency of a word is inversely proportional to the rank of the word, where rank 1 is given to the most frequent word, rank 2 to the second most frequent and so on. This is also called the **power law distribution**. If you plot the rank of a word versus its frequency of occurrence, you will get a graph as shown below.



The word with the highest frequency has the lowest rank. So, a point representing such a word will be closest to the y-axis to the top left. As the rank of a word decreases, its frequency of occurrence also decreases. The nature of the blue line in the graph suggests that the rank of a word and its occurrence frequency are inversely proportional, which gives the formula for Zipf's law as shown below.

f(word) × r(word) ~ constant

Be warned this is not a mathematically strong relationship, it weakly holds in the middle regions of the graph and does not hold at all at the extremes.

Tokenisation

An important question to ask while dealing with text data is: How do you extract features from messages so that they can be used to build an ML model? When you create an ML model, you need to feed in the features related to each message that the ML algorithm can use to build the model. But here, the data is text. As you know, ML works on numeric data, not text. Recall the case studies that you have gone through so far; the text columns are usually categorical data.

Since natural text is not categorical in nature, one cannot use the techniques used to precess categorical data to handle natural text. Unless the text is converted into a numerical form, the ML models cannot learn from the language data. How do you convert natural text into numerical form?

Consider the following sentences:

1. Cat ate the fish.
2. Raj ate his lunch.

One way to convert these sentences into numeric data is to create a dictionary of unique words and call them features as shown in the table provided below.

Features	cat	ate	the	fish	raj	his	lunch
Sentence_1	Present	Present	Present	Present	Absent	Absent	Absent
Sentence_2	Absent	Present	Absent	Absent	Present	Present	Present

How to represent the 'present's and the 'absent's in a numerical way to represent the sentence will be discussed later. For now, let's focus on the process of separating individual words from sentences. This process is called tokenisation.

Tokenisation is a technique that is used to split a text into smaller elements. These elements can be characters, words, sentences or even paragraphs, depending on the application that you are working on.

The guiding principle for the choice of tokens depends on the ability to extract maximum information from the text. The smaller the token, the more information you can extract; however, it will also increase the computational cost. On the other hand, larger tokens will be easy to compute; however, you would be able to extract less information.

The library that will be used in the Zomato case study is the Natural Language Toolkit (NLTK) library. In NLTK, you have different types of tokenisers that you can use in a variety of applications. The most popular tokenisers are as follows:

1. Word tokeniser, which splits text into different words
2. Sentence tokeniser, which splits text into different sentences
3. Tweet tokeniser, which handles emojis and hashtags that you see in social media texts
4. Regex tokeniser, which lets you build your own custom tokeniser using regex patterns of your choice

Stemming and Lemmatisation

Stemming and lemmatisation are other techniques that reduce the variety of words in a text corpus. Usually, any text contains a lot of different forms of the same word. Consider the piece of text provided below.

Person A: What did you eat for dinner today?

Person B: I ate a pizza.

The word ‘eat’ appears as ‘eat’ and ‘ate’ in the text. Although the information they contain is similar, they are different words. Similarly, a large corpus of words will have many variations of the same words, which results in high computation requirements.

Stemming is a rule-based technique that chops off the suffix of a word to obtain its root form, which is called the ‘stem’. For example, if you use a stemmer to stem the words of the string ‘The driver is racing his boss’ car’, the words ‘driver’ and ‘racing’ will be converted to their root form by just chopping off the suffixes ‘er’ and ‘ing’. So, ‘driver’ will be converted to ‘driv’ and ‘racing’ will be converted to ‘rac’.

You might think that the root forms (or stems) do not resemble the root words ‘drive’ and ‘race’. You need not worry about this because the stemmer will convert all the variants of ‘drive’ and ‘racing’ into the same root forms, i.e., it will convert ‘drive’, ‘driving’, etc. into ‘driv’ and ‘race’, ‘racer’, etc. into ‘rac’. Stemming gives satisfactory results in most cases, but it is not always accurate. This technique cannot be used in cases where the root form of a word cannot be derived by chopping off the suffix. In such cases, more sophisticated methods are needed. Later in the segment, you will learn the technique of lemmatisation, which helps resolve this issue.

Lemmatisation is a more sophisticated technique (and perhaps more 'intelligent') owing to the fact that it does not just chop off the suffix of a word. Instead, it takes an input word and searches for its base word by going recursively through all the variations of dictionary words. The base word in this case is called the ‘lemma’. Words such as ‘feet’, ‘drove’, ‘arose’ and ‘bought’ cannot be reduced to their correct base form using a stemmer. However, a lemmatiser can reduce them to their correct base form. The most popular lemmatiser is the WordNet lemmatiser, which was created by a team of researchers at Princeton University. You can read more about it here.

Nevertheless, you may sometimes find yourself confused as to whether to use a stemmer or a lemmatiser in your application. The following pointers may help you make this decision:

1. A stemmer is a rule-based technique, and hence, is much faster than a lemmatiser (which searches the dictionary to look for the lemma of a word). On the other hand, a stemmer typically gives less accurate results than a lemmatiser.
2. A lemmatiser is slower than a stemmer because of the dictionary lookup, but gives better results than a stemmer. Note that for a lemmatiser to perform accurately, you need to provide the part-of-speech tag of the input word (noun, verb, adjective, etc.). You will learn POS tagging in the next session. For now, it should suffice to know

that, often, there are cases when the POS tagger is quite inaccurate on your text and it worsens the performance of the lemmatiser as well. In short, you may want to consider a stemmer rather than a lemmatiser if you notice that the POS tagging is inaccurate.

Zomato Case Study

The objective of the Zomato case study was to use all data cleaning techniques on a real data set. The NLTK library was used in this case study.

Removing HTML tags

Beautiful Soup is a Python library that can be used to remove HTML tags. Passing the review to Beautiful Soup enables you to extract text. The code given below is the same code that Mahesh used to remove HTML tags.

```
#Removing the html strips
from bs4 import BeautifulSoup

def strip_html(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()

text = strip_html(text)
print(text)
```

A function is created from the Beautiful Soup library to extract text from HTML files. After that, every time you want to delete HTML tags, you can call this function.

Removing URLs

Regexes can be used to remove URLs. Every URL starts with a transfer protocol tag. In most cases, it is ‘https://’. You can use regex pattern matching to search for this pattern and remove all the text that follows it.

The code given below will show the implementation of regex to remove URLs.

```
import re

text = re.sub(r"http\S+", "", text)
print(text)
```

There is nothing new to see in this code; you have already learnt how to use regex. The above-mentioned code will match any text that starts with ‘http’ and replace it with blanks.

Removing Emojis

In the backend, all emojis and special characters are also encoded. Each of them has a specific pattern. You can use regex to create patterns that match the emojis and remove them using the ‘regrex_pattern.sub’ method. This method substitutes functions such as ‘find and replace’. Once a match pattern is detected, you can replace it with a blank.

Removing emojis is also a regex job, but one needs to know the right pattern to match. The code for this purpose is given below.

```
def deEmojify(text):
    regrex_pattern = re.compile(pattern = "["
        u"\U0001F600-\U0001F64F"  # emoticons
        u"\U0001F300-\U0001F5FF"  # symbols & pictographs
        u"\U0001F680-\U0001F6FF"  # transport & map symbols
        u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                           "]+", flags = re.UNICODE)
    return regrex_pattern.sub(r'',text)


text = deEmojify(text)
print(text)
```

The first command in the function is ‘regex.compile()’. It is used to create a regex object from the provided text.

Using the ASCII encoding, you can remove all the characters that are not present in the encoding, but it will result in loss of information. For instance, in this case, by removing the letter ‘pi’, the name of the dish being referred to was lost. Whether the loss of information is acceptable or not depends on the application that you are building. The code given below can help encode and decode text.

```
# ASCII encoding
```

```
text = text.encode('ascii', 'ignore')

print(text)


# Unicode encoding

def to_unicode(text):

    if isinstance(text, float):

        text = str(text)

    if isinstance(text, int):

        text = str(text)

    if not isinstance(text, str):

        text = text.decode('utf-8', 'ignore')

    return text
```

You already know the use of the encode and the decode functions. By executing the encode and decode cycle and suppressing the warnings, any character that is not present in the encoding standard is removed.

Next step was to convert all the letters into lowercase. The lower case conversion can be easily achieved using the '.lower()' function.

These steps were used to clean the text. The subsequent steps involve advanced techniques, such as stop word removal, stemming, lemmetisation and tokenisation.

The NLTK library has a predefined set of stop words. It can be directly used to build a function that can remove stop words. Given below is the pseudo code for removing the stop words from the reviews.

1. Separate the reviews into words.
   The 'toktok' tokeniser from the NLTK library is used to split the review sentences into a list of words.
2. From the list of words, check each word to see if it belongs to the list of stop words in the NLTK library.
3. If the word is present in the list of stop words, then do nothing; if not, add the word to a new list.
4. Combine the words from the new list to convert the list of words into a sentence.

Now, let’s compare the pseudo code to the actual code. The code will be much clearer and easier to understand now.

```
from nltk.corpus import stopwords
from nltk.tokenize.toktok import ToktokTokenizer

#Tokenization of text
tokenizer=ToktokTokenizer()

#Setting English stopwords
stopword_list=nltk.corpus.stopwords.words('english')

#Removing standard english stop words like prepositions, adverbs
from nltk.tokenize import word_tokenize,sent_tokenize

stop=set(stopwords.words('english'))
print(stop)
```

First, import the necessary tools and initialise them. Download and print the list of stop words.

```
#Removing the stopwords
def remove_stopwords(text, is_lower_case=False):
    # tokenize the test into a list of words.
    tokens = tokenizer.tokenize(text)
    # remove the empty spaces from right and left of the tokens.
    tokens = [token.strip() for token in tokens]
    # The second optional argument comes in the picture here.
    # If the option is true then the function will assume all the letters are in lower case
    if is_lower_case:
        # A new list is created with the words which are not in the list of stop words.
        filtered_tokens = [token for token in tokens if token not in stopword_list]
    else:
        # If the second argument is false then all token will also be converted to lowercase
        filtered_tokens = [token for token in tokens if token.lower() not in stopword_list]
    # join the text back together
    filtered_text = ' '.join(filtered_tokens)
    # return the text without stop words.
    return filtered_text

#Apply function on review column
zomato['Review']=zomato['Review'].apply(remove_stopwords)
```

Stemming and Lemmetisation

When you use stemming, you split the corpus into tokens and pass each token to the stemmer. The stemmer then determines the necessary modifications to the token and returns the stem of the token. Most of the time, the stems are not complete English words. This method serves its purpose of reducing unique words by converting all the words into their stems. The stemming code used in the demonstration is given below.

```
from nltk.stem import WordNetLemmatizer,SnowballStemmer
from nltk.stem.porter import PorterStemmer
nltk.download('wordnet')

def simple_stemmer(text):
    ps=SnowballStemmer(language='english')
    return ' '.join([ps.stem(word) for word in tokenizer.tokenize(text)])
```

In this demonstration, the ‘snowball’ stemmer has been used. The process of stemming is simple: You import the stemmer and pass each token, one by one, to the stemmer. Note that list comprehension has been used to make the code elegant and compact. The code after the return keyword serves three functions: It tokenises the provided text, passes each token to the stemmer and finally joins all the stems back together.

On the other hand, in lemmatisation, first, you pass the complete text to the POS tagger, and then, each word will have a POS tag attached to it. Now, you can tokenise the text and lemmatise it. For each word to be converted into its lemma, you need the word itself and its POS tag. Although the results obtained using a lemmatiser are more accurate, it uses significantly more computation resources. Let’s take a look at the code used for lemmatisation.

```
# Import all the necessary tools and packages.
# pos_tag is a module used to tag the part of speech.
from nltk.tag import pos_tag
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
#Lemmatizer example
def lemmatize_all(sentence):
    wnl = WordNetLemmatizer()
    # create a loop which processes each token at a time.
    # During each loop one word is first tagged by pos_tag
    # The same tag is converted to a form which the lemmatizer understands.
    for word, tag in pos_tag(word_tokenize(sentence)):
        if tag.startswith("NN"):
            yield wnl.lemmatize(word, pos='n')
        elif tag.startswith('VB'):
            yield wnl.lemmatize(word, pos='v')
        elif tag.startswith('JJ'):
            yield wnl.lemmatize(word, pos='a')
        # if none of the tags match, the word is returned as is.
        else:
            yield word

def lemmatize_text(text):
    return ' '.join(lemmatize_all(text))
```

The above-mentioned code is a bit complex to understand. First, all the necessary functions are imported. Again, let's focus on the ‘lemmatize_all’ function. The input to the function is a tokenised sentence with each token (word, in this case) tagged with its relevant POS, such as noun, verb, adverb or preposition. This tagging is done by another module in the NLTK library, the ‘pos_tag’. Then, the nested ‘if statement’ makes sure that the tags are understandable by the ‘WordNetLemmatizer()’ function. If the POS tag does not match any of the wordnet-specified parts of speech, then the word is returned as is.

Summary

Advanced Lexical Processing

Bag of Words (BoW) Model

Any MLalgorithm works on matrices made of digits and not text. Feature extraction algorithms are responsible for converting a clean text into digits.

Consider the text corpus given below. It shows text in each document along with its sentiment label. A document in a text corpus is an entity that corresponds to one complete row in the feature matrix. For instance, in the Zomato review dataset, a document corresponded to one entire review. It could comprise multiple sentences or just a few words. For analytical purposes, one review represents one entity and it is referred to as a document.

	Phone review	**Sentiment**
Doc_1	camera, not, good	Negative
Doc_2	screen, awesome, microprocessor, slow, happy	Positive
DOc_3	hotspot, not, connect	Negative

Notice the text in the given documents. The text is clean and has no noise, which implies that it is already tokenised and lemmetised and all the stop words have been removed. If you were to convert such a corpus into features, you would arrange all the unique words in the corpus column-wise and each document would be represented row-wise as shown in the matrix below.

	camera	not	good	screen	awesome	microprocessor	slow	happy	hotspot	connect	...	sentiment
doc_1	value	value	value	0	0	0	0	0	0	0	0	negative
doc_2	0	0	0	value	value	value	value	value	0	0	0	positive
doc_3	0	value	0	0	0	value	0	0	0	value	0	negative

Next, you need to learn how to fill the matrix. In each row, you mark zero in the columns corresponding to the words that do not appear in the document. And you fill a value in the columns for words that are present in the document.

The values that you fill in the columns can be determined through multiple techniques. In this session, you will learn two such techniques, which are as follows:

1. Bag of words (BoW)
2. Term frequency inverse document frequency (TFIDF)

Bag of Words (BoW)

In the BoW technique, the value in each cell is just the count of words that are present in that document.

The BoW model uses the frequency of occurrence of a word in a document to fill up the features matrix. To attain a better understanding of how BoW works, let's try to create a feature matrix for the two documents given below.

Doc 1: quick, brown fox, jump, over, lazy, dog, back

Doc 2: now, time, all good, men, come, aid, their, party

Again, note that, here, clean tokenised text is being used. Usually, it will take a lot of effort to bring raw data to this state. To create a feature matrix, represent all the unique words in these documents as separate columns and the documents as rows.

	aid	all	brown	come	dog	fox	good	jump	lazy	man	now	over	party	quick	their	time
Document1																
Document2																

The value in each cell will be the frequency of occurrence of the corresponding word in that document. For example, in the first row and the first column, 0 will be entered because the word "aid" does not appear in Document 1. All other cells will be filled in a similar way. The filled matrix is shown below.

	aid	all	brown	come	dog	fox	good	jump	lazy	man	now	over	party	quick	their	time
Document1	0	0	1	0	1	1	0	1	1	0	0	1	0	1	0	0
Document2	1	1	0	1	0	0	1	0	0	1	1	0	1	0	1	1

So, Document 1 can be represented as [0010110110010100]. There will be similar vectors for all other documents. The length of count vectors for all the documents is the same because the number of columns for each row will be the same. It is not necessary that all count vectors will only have 0s and 1s. If a word repeats more than once in a sentence, then the corresponding cell will harbor the actual frequency of occurrence of that word.

TFIDF Representation

The TFIDF representation, also called the TFIDF model, takes into account the importance of each word in the given documents. In the BoW model, each word is assumed to be equally important in the corpus of documents, which, of course, is not correct.

Given below is the formula to calculate the TFIDF weight of a term in a document:

$$tf_{t,d} = \frac{frequency\ of\ term\ 't'\ in\ document\ 'd'}{total\ terms\ in\ document\ 'd'}$$

$$idf_t = log\frac{total\ number\ of\ documents}{total\ documents\ that\ have\ the\ term\ 't'}$$

The log in the above formula is with base 10. Now, the TFIDF score for any term in a document is the product of the two variables determined using the above-mentioned formulas:

$$tf - idf = tf_{t,d} * idf_t$$

Higher weights are assigned to terms that occur frequently in one document and are rare among all other documents. On the other hand, a low score is assigned to the terms that occur commonly across all documents.

The TFIDF model intuitively assigns higher value to the words that are rare. The term frequency is higher for the words that occur frequently in a document. Whereas, the inverse document frequency is higher for the terms that are rare. As a result, only the more significant and rare words get assigned high numerical values.

Sentiment Analysis Case Study

The objective of this study was to build a model to predict the sentiment of a statement. The first step of the case study was to clean the data. Basic lexical techniques, such as removing punctuations, etc., can be performed using the following function.

```
#Collating all functions together and applying them for the 'IMDb reviews' dataset
def strip_html(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()

#Removing emojis
def deEmojify(text):
    regrex_pattern = re.compile(pattern = "["
        u"\U0001F600-\U0001F64F"  # emoticons
        u"\U0001F300-\U0001F5FF"  # symbols & pictographs
        u"\U0001F680-\U0001F6FF"  # transport & map symbols
        u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                           "]+", flags = re.UNICODE)
    return regrex_pattern.sub(r'',text)

#Text-encoding: UTF-8 encoder
def to_unicode(text):
    if isinstance(text, float):
        text = str(text)
    if isinstance(text, int):
        text = str(text)
    if not isinstance(text, str):
        text = text.decode('utf-8', 'ignore')
    return text

#Removing the square brackets
def remove_between_square_brackets(text):
    return re.sub('\[[^]]*\]', '', text)


#Define function for removing special characters
def remove_special_characters(text, remove_digits=True):
    pattern=r'[^a-zA-z0-9\s]'
    text=re.sub(pattern,'',text)
    return text

#Removing the noisy text
def denoise_text(text):
    text = to_unicode(text)
    text = strip_html(text)
    text = re.sub(r"http\S+", "", text)
    text = deEmojify(text)
    text = text.encode('ascii', 'ignore')
    text = to_unicode(text)
    text = remove_between_square_brackets(text)
    text = remove_special_characters(text)
    text = text.lower()
    return text
```

Next, stemming and lemmatisation was used on one specific review to compare performance.

Difference Between the Computation Effort in Stemming and Lemmatisation

Time taken by the ‘Snowball’ stemmer to stem all the words in this review is about 2.19 ms. Whereas, the time taken by the ‘WordNet’ lemmatiser to find the lemma for all the words in the same review is 9.99 sec. All the other variables, such as the performance of the machine, are the same in both cases. The change in the execution time is only attributed to the fact that lemmatisation requires more computation than stemming.

Apart from the execution time, observe the words in the outputs of the stemmer and the lemmatiser. Stemmer produces incomplete words, whereas lemmatiser produces complete words.

The decision to use a stemmer or a lemmatiser is based on the computational resources available. If enough computational resources are available, a lemmatiser is preferred, and if the text corpus is extremely large and the resources are not enough, then a stemmer is preferred.

Sentiment Analysis: BoW Model

LabelBinarizer

Recall the IMDB dataset. It had two columns: the reviews and the labels for each review, that is, either ‘positive’ or ‘negative’. Even the labels need to be in numerical form for the logistic algorithm to process them. The label column is categorical and comprises only two classes. So, it is quite easy to convert the categorical data into numerical data. To this end, the code given below was used in this demonstration.

```
from sklearn.preprocessing import LabelBinarizer

#Labelling the sentient data
lb=LabelBinarizer()

#Transformed sentiment data
sentiment_data=lb.fit_transform(imdb['sentiment'])
print(sentiment_data.shape)
```

‘LabelBinarizer’ is a function that converts categorical labels into numerical labels. You simply import the function and apply it to the relevant column.

Extracting Features

Now, to convert the clean IMDB data into features, you use a function called the ‘CountVectorizer’. Like most of the other prebuilt algorithms, there are two different stages in the application: First, fit the algorithm, and then, transform the data. The ‘fit’ and the ‘transform’ stages are more complex in case of text data compared to normal data.

The ‘fit’ command is executed using the code shown below.

```
from sklearn.feature_extraction.text import CountVectorizer

#Creating a matrix with reviews in row and unique words as columns
# and frequency of word in review as values.
#Count vectorizer for bag of words
cv=CountVectorizer()

#Fitting model on entire data
cv_fit = cv.fit(norm_reviews)
```

Recall the feature matrix. The columns in the feature matrix are unique words in the corpus and the rows are count vectors describing specific documents. When the ‘fit’ command is executed, all the unique words from the entire corpus are collected together. The above-mentioned code executes the ‘fit’ command. Next comes the ‘transform’ command, and here, the complexity begins. Let’s first take a look at the code to implement the ‘transform’ command.

```
#Normalised train reviews
norm_train_reviews=imdb.review[:8000]
print('train:','\n',norm_train_reviews[0])
norm_train_cv_reviews=cv_fit.transform(norm_train_reviews)

#Normalised test reviews
norm_test_reviews=imdb.review[8000:]
print('test:','\n',norm_test_reviews[8001])
norm_test_cv_reviews=cv_fit.transform(norm_test_reviews)
```

Let’s summarise the ‘fit’ and the ‘transform’ processes.

1. Fit the BoW model on the entire text corpus.
2. Split the data into ‘train’ and ‘test’.
3. Independently transform the ‘train’ and the ‘test’ datasets.

Why is the ‘fit’ operation performed first, and then, the ‘transform’ operation is applied independently on the ‘train’ and the ‘test’ datasets?

Let’s take a look at a very simple example to understand the answer to this question.

Consider that you have a dataset of text as shown below.

Document 1: Word_1 Word_2

Document 2: Word_2 Word_3

Document 3: Word_1 Word_2 Word_4

When the 'fit' command is executed on the entire dataset, the feature matrix will contain columns corresponding to all four unique words. Thus, there will be four columns in the feature matrix.

Next, the data is split into 'train' and 'test' datasets. In this case, let documents 1 and 2 represent the training data. Given below is the feature matrix with the training data:

	Word_1	Word_2	Word_3	Word_4
Doc1	1	1	0	0
Doc2	0	1	1	0

Then, the 'transform' command picks up each document, cross-checks the tokens in the document with the feature list and populates the row with the right values. **So, by executing the 'fit' command on the entire dataset, you have made sure that there are four values in each row corresponding to each word.**

Even if 'Word_4' is exclusively present in the 'test' dataset, and if all the values in that column are 0, by keeping the words unique to the 'test' data in the 'fit' command, you have ensured that the length of the count vector remains the same. But, why is it necessary to keep the length of the count vector the same for all documents?

Recall the ML algorithms you have learnt. The actual computations occur on matrix equations. So, all rows in the matrix need to have the same shape. If the rows have inconsistent shapes, the matrix operations will not work.

Hence, by creating the feature matrix such that it contains all the unique words from the given text corpus, you are, in a way, ensuring that the length of the feature matrix stays the same. This is the reason for executing the 'fit' command on the entire data but executing the 'transform' command only on the 'training' set.

Next, you will also need to separate the labels for each document. The code given below will split the labels.

```
#Splitting the sentiment data
train_sentiments=sentiment_data[:8000]
test_sentiments=sentiment_data[8000:]
```

Here, simple series indexing is used to split the data. Now that you have the ‘train’ and the ‘test’ sets, you are ready to build a model. Through the code given below, let’s take a look at the model building process.

```
from sklearn.linear_model import LogisticRegression,SGDClassifier

#Training the model
lr=LogisticRegression(penalty='l2',max_iter=500,C=1,random_state=42)

#Fitting the model for the bag of words
lr_bow=lr.fit(norm_train_cv_reviews,train_sentiments)
print(lr_bow)
```

As you can observe in the above-mentioned code, a simple logistic regression model with L2 regularisation is used to build the classification engine. Then, you use this model to make predictions for the ‘test’ dataset.

```
#Predicting the model for bag of words
lr_bow_predict=lr.predict(norm_test_cv_reviews)
print(lr_bow_predict)
```

You are already familiar with this code, there is nothing new to learn.

Sentiment Analysis: TFIDF Model

The overall TFIDF model building process was exactly similar to the process used to build the BoW model.

1. Extract the features.
2. Split the ‘train’ and the ‘test’ data features.
3. Split the labels.
4. Train the model.
5. Use the model for prediction.

The code used to build the TFIDF model is given below.

```
#Transformed train reviews
norm_reviews=imdb.review
```

```
#Term-frequencey * inverse document frequency matrix
from sklearn.feature_extraction.text import TfidfVectorizer

#Applying TF-IDF vectorizer
tv=TfidfVectorizer()

#Fitting model on entire data
tv_fit = tv.fit(norm_reviews)

#Normalised train reviews
norm_train_reviews=imdb.review[:8000]
print('train:','\n',norm_train_reviews[0])
norm_train_tv_reviews=tv_fit.transform(norm_train_reviews)

#Normalised test reviews
norm_test_reviews=imdb.review[8000:]
print('test:','\n',norm_test_reviews[8001])
norm_test_tv_reviews=tv_fit.transform(norm_test_reviews)

norm_train_tv_reviews.shape

#Splitting the sentiment data
train_sentiments=sentiment_data[:8000]
test_sentiments=sentiment_data[8000:]

from sklearn.linear_model import LogisticRegression,SGDClassifier

#Training the model
lr=LogisticRegression(penalty='l2',max_iter=500,C=1,random_state=42)

#Fitting the model for tf-idf features
lr_tfidf=lr.fit(norm_train_tv_reviews,train_sentiments)
print(lr_tfidf)

#Predicting the model for tf-idf features
lr_tfidf_predict=lr.predict(norm_test_tv_reviews)
print(lr_tfidf_predict)
```

On comparing both the logistic models, it seems that their predictability is very close to each other. Let’s take a look at the code used to determine the predictive accuracy of the two models.

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

#Accuracy score for bag of words
lr_bow_score=accuracy_score(test_sentiments,lr_bow_predict)
print("lr_bow_score :",lr_bow_score)

#Accuracy score for tf idf features
```

```
lr_tfidf_score=accuracy_score(test_sentiments,lr_tfidf_predict)
print("lr_tfidf_score :",lr_tfidf_score)
```

So, comparing the accuracy of the models will give you a fair comparison of the performance of both models.

lr_bow_score	0.8654
lr_tfidf_score	0.8675

The accuracy score for both the models is the same upto two decimal places.

Next, all the other evaluation matrices, such as recall, precision, etc., were also compared.

```
#Classification report for bag of words

lr_bow_report=classification_report(test_sentiments,lr_bow_predict,target_names=['Positiv
e','Negative'])

print(lr_bow_report)

#Classification report for tf idf features

lr_tfidf_report=classification_report(test_sentiments,lr_tfidf_predict,target_names=['Positiv
e','Negative'])

print(lr_tfidf_report)
```

Again, most of the assessed metrics for both models were almost similar with minor variations.

```
              precision    recall  f1-score   support

    Positive       0.87      0.85      0.86       975
    Negative       0.86      0.88      0.87      1025

    accuracy                           0.86      2000
   macro avg       0.86      0.86      0.86      2000
weighted avg       0.86      0.86      0.86      2000
              precision    recall  f1-score   support

    Positive       0.89      0.83      0.86       975
    Negative       0.85      0.90      0.87      1025

    accuracy                           0.87      2000
   macro avg       0.87      0.87      0.87      2000
weighted avg       0.87      0.87      0.87      2000
```

Sentiment Analysis: TextBlob Library

The TextBlob library is a pre-trained language model that can be used to classify text on the basis of sentiments. The training is done and the coefficients for each unique token are stored in the library.

```
from textblob import TextBlob

  #Create a function to get the polarity

def getPolarity(text):

    return TextBlob(text).sentiment.polarity

def getAnalysis(score):

    if score < 0:

        return "negative"

    else:

        return "positive"

imdb["textblob"] = imdb["review"].apply(getPolarity)

imdb["textblob_flag"] = imdb["textblob"].apply(getAnalysis)
```

By executing the above-mentioned code, you are using the pre-trained TextBlob model to predict the sentiment of the review. TextBlob is a language model based on neural networks. Going deep into the working of textbooks is out of the scope of this module. Let's focus on its execution. It takes a document as input and assigns a sentiment score to it. Then, a threshold of 0 is set. If the score is above 0, the review is marked as positive, else it is marked as negative. Quite simple, right?

Let's now take a look at the performance of the TextBlob model when implemented for the IMDB data.

```
#Classification report predictions from TextBlob
```

```
textblob_report=classification_report(test_sentiments,imdb["textblob_flag"][8000:],target_names=['Positive','Negative'])

print(textblob_report)
```

To make an ‘apples to apples’ comparison, you have used the same set of reviews ‘8001 to 10000’. Given below are the evaluation metrics of the TextBlob model.

```
              precision    recall  f1-score   support

    Positive       0.79      0.39      0.52       975
    Negative       0.61      0.90      0.73      1025

    accuracy                           0.65      2000
   macro avg       0.70      0.65      0.62      2000
weighted avg       0.70      0.65      0.63      2000
```

Compared to the TFIDF model, the TextBlob model has poorer performance with respect to almost all the metrics. If that is the case, then why do you need to bother with such models?

The answer is ‘generalisability’. The TextBlob model is generalisable. You need to understand that the TFIDF model was explicitly trained on the IMDB dataset. Suppose you use the same TFIDF model on a sports tweets dataset, what do you think will be the model performance? Most probably, in this case, the TextBlob will perform well, because there will be many tokens that the TFIDF model will not have even seen before.

Cosine Similarity

The cosine similarity model is used to find similar documents from a corpus.Consider the following set of documents.

D1: *‘Interstellar is a science based movie. It uses physics concepts’*

D2: *‘Gravitation is a physics concept. It is one of the important fields in science and there is a movie Gravity based on this’*

D3: *‘Fast and Furious is a thriller movie. It is based on cars and adventures’*

The first step is to find the TFIDF matrix for these sentences.

	interstellar	science	movie	physics	concepts	gravitations	fields	gravity	fast	furious	thriller	car	adventures
D1	0.53	0.40	0.31	0.40	0.53	0	0	0	0	0	0	0	0
D2	0	0.32	0.25	0.42	0.42	0.42	0.42	0.32	0	0	0	0	0
D3	0	0	0.43	0	0	0	0	0	0.43	0.43	0.43	0.43	0.25

Now, you use the TFIDF values of the words that are common to both the documents to find the cosine similarity score. Given below is the formula to determine the cosine similarity.

$$cos\theta = \frac{X \times Y}{|X||Y|}$$

Here, X and Y are vectors of the TFIDF values, corresponding to common words in the two documents. Then, this formula is used to find similarity between D1 and D2.

1. The common values in D1 and D2 are highlighted.

	interstellar	science	movie	physics	concepts	gravitations	fields	gravity	fast	furious	thriller	car	adventures
D1	0.53	**0.40**	**0.31**	**0.40**	**0.53**	0	0	0	0	0	0	0	0
D2	0	**0.32**	**0.25**	**0.42**	**0.42**	0.42	0.42	0.32	0	0	0	0	0
D3	0	0	0.43	0	0	0	0	0	0.43	0.43	0.43	0.43	0.25

2. Now you apply the cosine similarity formula to the four highlighted sets of TFIDF values.

$$\text{Cosine Similarity} := \frac{(0.40\times0.32)+(0.31\times0.25)+(0.40\times0.42)+(0.53\times0.42)}{\sqrt{0.40^2+0.31^2+0.40^2+0.53^2}\times\sqrt{0.32^2+0.25^2+0.42^2+0.42^2}}$$

3. So, the cosine similarity between D1 and D2 = 0.992.

If the value of cosine similarity is closer to 0, the vectors of these two documents are approximately at 90 degrees to each other and they are not at all similar to each other.

If the value of cos similarity is closer to 1, the vectors of these two documents are parallel to each other and the documents are highly similar.

In the given case, documents D1 and D2 are very similar to each other, as the cosine similarity value is 0.992, which is close to 1.

Conversely, the cosine distance (1 - cosine similarity) value is quite high between D1 and D2. Contrary to cosine similarity, which signifies the degree of similarity between the documents, cosine distance signifies the degree of difference between a pair of documents.

Cosine Similarity: Demonstration

First step is to clean the data. Then, fInd the TFIDF matrix for the entire dataset. The code for creating a TFIDF matrix is given below.

```
# Initialize an instance of tf-idf Vectorizer
tfidf_vectorizer = TfidfVectorizer()

# Generate the tf-idf vectors for the corpus
tfidf_matrix = tfidf_vectorizer.fit_transform(corpus)
```

The output of the above-mentioned code is the TFIDF matrix shown below.

```
   architecture      body     brown  ...     solar       sun    system
0      0.000000  0.458815  0.000000  ...  0.370169  0.307274  0.370169
1      0.000000  0.000000  0.000000  ...  0.336446  0.279281  0.336446
2      0.000000  0.000000  0.000000  ...  0.000000  0.380406  0.000000
3      0.523358  0.000000  0.000000  ...  0.000000  0.000000  0.000000
4      0.000000  0.000000  0.408248  ...  0.000000  0.000000  0.000000
```

Using this TFIDF matrix, you can then find the cosine similarity scores. The cosine similarity function is present in the 'sklearn' library. You can import the library and use it to calculate the cosine similarity for the declared corpus.

The code for calculating the cosine similarity is given below.

```
cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)
print(cosine_sim)
```

From this TFIDF matrix, you can then find the cosine similarity scores. Each row and each column in the above-mentioned matrix represent a document. This matrix shows the relationship between each pair of documents.

n-gram Models

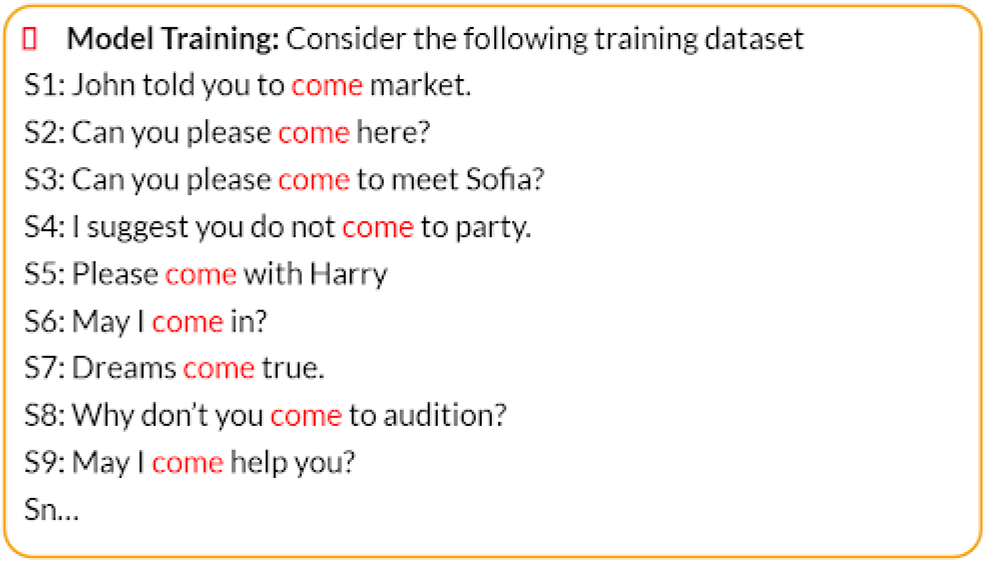
These models use probability distributions to determine the word that is most likely to come after the last word you have typed. Similar to all other ML models, the execution of this model is also split into two parts, which are as follows:

1. Model training using the 'training' dataset
2. Prediction and testing using the 'test' dataset

upGrad

Training, in this case, comprises calculation of the probability of each word given the preceding word.

Suppose the last word you typed in the text was ‘come’. Now, you need to search the entire corpus of text and find the word ‘come’.



▯ **Model Training:** Consider the following training dataset

S1: John told you to come market.

S2: Can you please come here?

S3: Can you please come to meet Sofia?

S4: I suggest you do not come to party.

S5: Please come with Harry

S6: May I come in?

S7: Dreams come true.

S8: Why don't you come to audition?

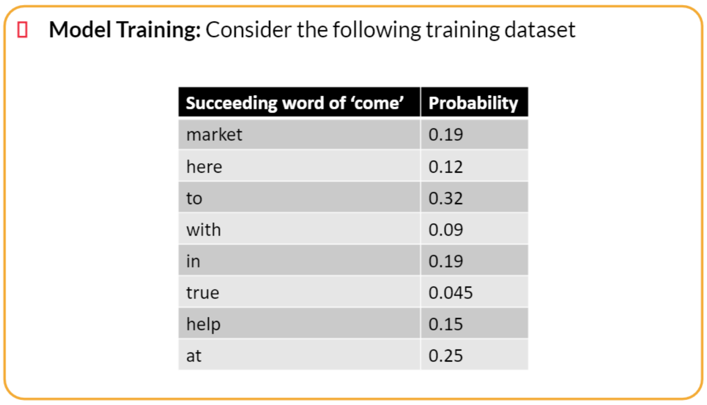
S9: May I come help you?

Sn...

Next, you need to find the probability of occurrence of each word that appears after ‘come’. This probability can be determined by taking the ratio of the total number of times a particular word appears after ‘come’ to the total number of documents that have the word ‘come’. For instance, in the given corpus, the probability of the word ‘to’ to appear after the word ‘come’ is as follows:

Probability = 2/9, considering the 9 documents shown above

Similarly, if you repeat the exercise for the rest of the words, you will obtain a probability distribution, as shown below:

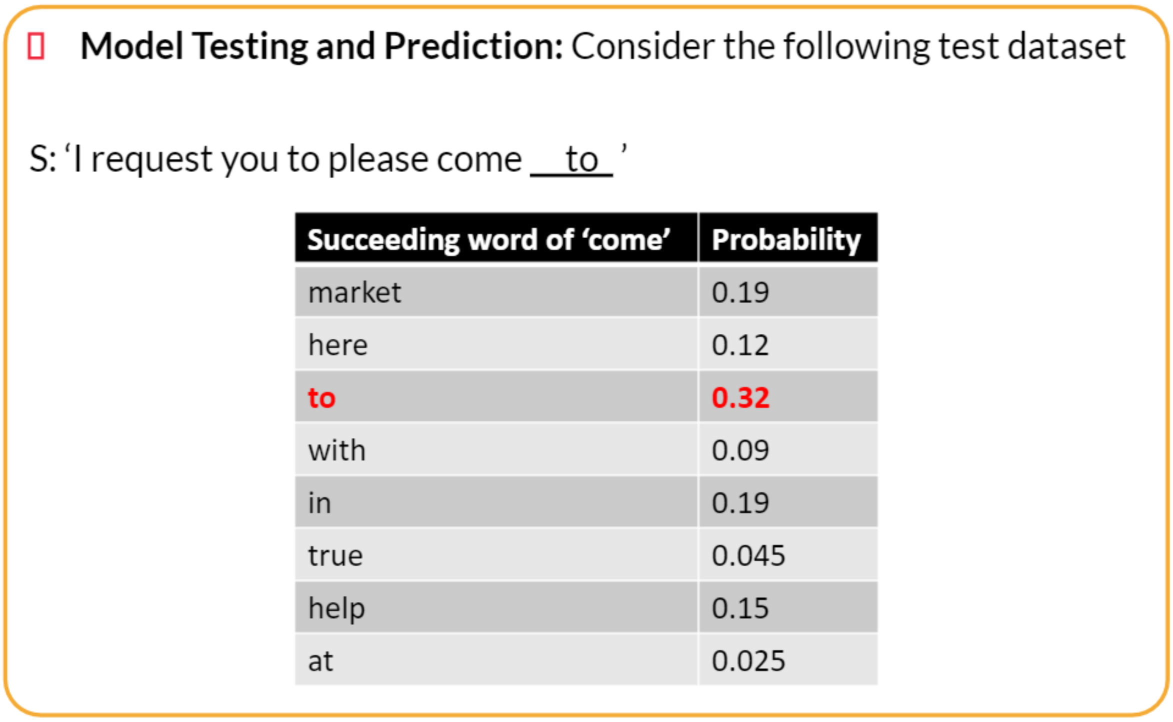


▯ **Model Training:** Consider the following training dataset

Succeeding word of ‘come’	Probability
market	0.19
here	0.12
to	0.32
with	0.09
in	0.19
true	0.045
help	0.15
at	0.25

Now, you have obtained a probability distribution for all the words available in the corpus. The next step is to make some predictions that can be shown to the user.

You recommend the most probable word. If you need to recommend three words, you recommend the top three most likely words.



Model Testing and Prediction: Consider the following test dataset

S: 'I request you to please come __to__ '

Succeeding word of 'come'	Probability
market	0.19
here	0.12
to	**0.32**
with	0.09
in	0.19
true	0.045
help	0.15
at	0.025

Finally, there is one variation. Till this point, you have learnt how the recommendation process works when a single word is being considered to predict the next word. What if you want to take more than one preceding word into consideration?

The process of recommending subsequent words taking into account two or even three preceding words is exactly the same as it is while taking into account one preceding word. This process is as follows:

1. Find all the documents with the given sequence of words.
2. Calculate the probability distribution of all the succeeding words.
3. Recommend the words with the maximum probability.

If only one preceding word is considered, it is called the unigram model; for two words, it is called the bigram model; for three words, it is called the trigram model; and so on. For 'n' number of preceding words, it is called the **n-gram** model.