**Contents**

1. [**Text Encoding**](#text_encoding)
2. [**Word Frequencies and stop words**](#word_frequencies_and_stop_words)
3. [**Tokenization**](#tokenization)
4. [**Bag-of-Words Representation**](#bag_of_words_representation)
5. [**Stemming and Lemmatization**](#stemming_and_lemmatization)
6. [**TF-IDF Representation**](#tfidf_representation)
7. [**Canonicalization**](#canonicalization)
8. [**Phonetic Hashing**](#phonetic_hashing)
9. [**Pointwise Mutual Information**](#pointwise_mutual_information)

**Lexical Processing**

* Lexical Processing: First, we will just convert the raw text into words and, depending on our application's needs, into sentences or paragraphs as well.
  + 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.
  + 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 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.
  + For a simple application like spam detection, lexical processing works just fine, but it is usually not enough in more complex applications, like, say, machine translation. 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. Hence, we need a more advanced system of analysis.
* Syntactic Processing: So, the next step after 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.
  + 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.
  + There are various other ways in which these syntactic analyses can help us enhance our understanding. For example, a question answering system where we ask a 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 an 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 give an 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.
  + 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 we can assume that the meanings of the two words are similar as well.
  + 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”. We can probably save these relations manually, but it will help us a lot more, if we can train our machine to look for the relations on its own, and learn them.
* Once we have the meaning of the words, obtained via semantic analysis, we 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 semantic 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, we'll learn how text is stored on machines.

**Text Encoding**

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

Now, it is not necessary that when we work with text, we will 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 us to work with non-English text, we 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.

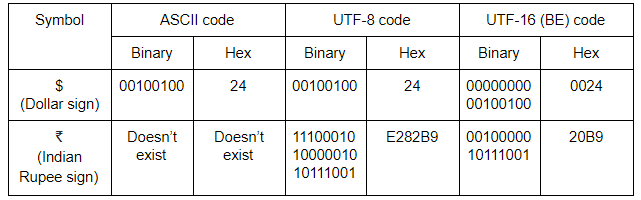
When ASCII was built, English alphabets were the only alphabets that were present on the keyboard. With time, new languages began to show up on keyboard sets which brought new characters. ASCII became outdated and couldn’t incorporate so many languages. A new standard has come into existence in recent 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, we need to know what kind of encoding the text has and if required, modify it to another encoding format.

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.



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. We can also look at [this](https://mothereff.in/utf-8) UTF-8 encoder-decoder to look how a string is stored. Note that, the online tool gives us 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**')

Word Frequencies and Stop Words

While working with any kind of data, the first step that we usually do is to explore and understand it better. In order to explore text data, we need to do some basic preprocessing steps.

Now, a text is made of characters, words, sentences and paragraphs. The most basic statistical analysis you can do is to look at the **word frequency distribution,**i.e. visualising the word frequencies of a given text corpus.

It turns out that there is a common pattern you see when we plot word frequencies in a fairly large corpus of text, such as a corpus of news articles, user reviews, Wikipedia articles, etc.

The[**Zipf's law**](https://en.wikipedia.org/wiki/Zipf%27s_law)(discovered by the 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, 2 to the second most frequent and so on. This is also called the **power law distribution.**

The Zipf's law helps us form the basic intuition for **stopwords -**these are the words having the highest frequencies (or lowest ranks) in the text, and are typically of limited 'importance'.

Broadly, there are three kinds of words present in any text corpus:

* Highly frequent words, called stop words, such as ‘is’, ‘an’, ‘the’, etc.
* Significant words, which are typically more important to understand the text
* Rarely occurring words, which are again less important than significant words

Generally speaking, stopwords are removed from the text for two reasons:

1. They provide no useful information, especially in applications such as spam detector or search engine. Therefore, we’re going to remove stopwords from the spam dataset.
2. Since the frequency of words is very high, removing stopwords results in a much smaller data as far as the size of data is concerned. Reduced size results in faster computation on text data. There’s also the advantage of less number of features to deal with if stopwords are removed.

However, there are exceptions when these words should not be removed such as POS (parts of speech) tagging and parsing where stopwords are preserved because they provide meaningful (grammatical) information in those applications. Generally, stopwords are removed unless they prove to be very helpful in our application or analysis.

On the other hand, we’re not going to remove the rarely occurring words because they might provide useful information in spam detection. Also, removing them provides no added efficiency in computation since their frequency is so low.

**Tokenization**

There is another thing to think about - how to extract features from the messages so that they could be used to build a classifier. When we create any machine learning model such as a spam detector, we will need to feed in features related to each message that the machine learning algorithm can take in and build the model. But here, in the spam dataset, we only have two columns - one column contains the message and the other contains the label related to the message. And as we know, machine learning works on numeric data, not text. Earlier when we worked with text columns, we either treated them as categorical variables and converted each categorical variable to numeric variable by either assigning numeric values to each category, or created dummy variables. Here, neither of these, since the message column is unique, it’s not a categorical variable. If you treat it as a category, our model will fail miserably.

To deal with this problem, we will extract features from the messages. From each message we’ll extract each word by breaking each message into separate words or 'tokens'.

This technique is called **tokenisation** - a technique that’s used to split the text into smaller elements. These elements can be characters, words, sentences, or even paragraphs depending on the application we’re working on.

In the spam detector case, we will break each message into different words, so it’s called **word tokenisation**. Similarly, we have other types of tokenisation techniques such as character tokenisation, sentence tokenisation, etc. Different types of tokenisation are needed in different scenarios.

There are multiple ways of doing a particular thing in Python. To tokenise words, we can use the split() method that just splits text on white spaces, by default. This method doesn’t always give good results. You are better off using NLTK’s tokeniser which handles various complexities of text. One of them is that it handles**contractions** such as “can’t”, “hasn’t”, “wouldn’t”, and other contraction words and splits these up although there is no space between them. On the other hand, it is smart enough to not split words such as “o’clock” which is not a contraction word.

In NLTK, you also have different types of tokenisers present that you can use in different applications. The most popular tokenisers are:

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

**Bag-of-Words Representation**

The most common and most popular approach is to create a **bag-of-words representation** of the text data that we have. The central idea is that any given piece of text, i.e., tweets, articles, messages, emails etc., can be “represented” by a list of all the words that occur in it (after removing the stopwords), where the **sequence of occurrence does not matter**. You can visualise it as the “bag” of all “words” that occur in it. For example, consider the messages:

“Gangs of Wasseypur is a great movie”

The bag of words representation for this message would be:



This way, you can create “bags” for representing each of the messages in your training and test data set. But how do you go from these bags to building a spam classifier?

Let’s say the bags, for most of the spam messages, contain words such as prize, lottery etc., and most of the ham bags don’t. Now, whenever you run into a new message, just look at its “bag-of-words” representation. Does the bag for this message resemble that of messages you already know as spam, or does it not resemble them? Based on the answer to the previous question, you can then classify the message.

Now, the next question is, how do you get a machine to do all of that? Well, turns out that for doing that, you need to represent all the bags in a matrix format, after which you can use ML algorithms such as naive Bayes, logistic regression, SVM etc., to do the final classification.

So, that’s how text is represented in the form of matrix. It can then be used to train machine learning models. Each document sits on a separate row and each word of the vocabulary has a its own column. These vocabulary words are also called as **features** of the text.

The bag-of-words representation is also called bag-of-words model but this is not to be confused with a machine learning model. A bag-of-words model is just the matrix that you get from text data.

Another thing to note is that the values inside any cell can be filled in two ways - 1) you can either fill the cell with the frequency of a word (i.e. a cell can have a value of 0 or more), or 2) fill the cell with either 0, in case the word is not present or 1.

Both approaches work fine and don’t usually result in a big difference. The frequency approach is slightly more popular and the NLTK library in Python also fills the bag-of-words model with word frequencies rather than binary 0 or 1 values.

During the model building, there might be some features such as ‘get’ and ‘getting’, ‘goes’ and ‘going’, ‘see’ and ‘seeing’ and along with a lot of other duplicate features. They are not exactly duplicates but they’re redundant in the sense that they’re not giving you any extra information about the message. In fact, the words ‘winner’ and ‘win’ are equivalent when your goal is to detect whether a message is spam or not.

Hence, keeping the two separate is actually going to hinder the performance of the machine learning algorithm since it is redundant information. Also, this redundancy is going to increase the number of features due to which the classifier can face the *curse of dimensionality*(error increases with the increase in number of features). To get rid of this problem, we’re going to learn two more preprocessing techniques - **stemming** and **lemmatization**

**Stemming and Lemmatization**

If you noticed, the repeated tokens or features were nothing but a variation or an **inflected form**of the other token. For example, the word ‘seeing’ is an inflection of the word ‘see’. Similarly, the word ‘limited’ is an inflection of the word ‘limit’. The two techniques that you just learnt reduce these inflected words to the original base form. But which is one is a better technique in what situations? Let’s look at them one by one:

**Stemming**

It is a **rule-based** technique that just chops off the suffix of a word to get 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 in his boss’ car", the words ‘driver’ and ‘racing’ will be converted to their root form by just chopping of 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) don’t resemble the root words - ‘drive’ and ‘race’. You don’t have to worry about this because the stemmer will convert all the variants of ‘drive’ and ‘racing’ to those root forms only. So, it will convert ‘drive’, ‘driving’, etc. to ‘driv’, and ‘race’, ‘racer’, etc. to ‘rac’. This gives us satisfactory results in most cases.

There are two popular stemmers:

* **Porter stemmer**: This was developed in 1980 and works only on English words. You can find all the detailed rules of this stemmer [here](http://snowball.tartarus.org/algorithms/porter/stemmer.html).
* **Snowball stemmer**: This is a more versatile stemmer that not only works on English words but also on words of other languages such as French, German, Italian, Finnish, Russian, and many more languages. You can learn more about this stemmer [here](http://snowball.tartarus.org/).

**Lemmatization**

This is a more sophisticated technique (and perhaps more 'intelligent') in the sense that it doesn’t 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’, ‘bought’, etc. can’t be reduced to their correct base form using a stemmer. But a lemmatizer can reduce them to their correct base form. The most popular lemmatizer is the **WordNet lemmatizer** created by a team of researchers at the Princeton university. You can read more about it [here](https://wordnet.princeton.edu/).

Nevertheless, you may sometimes find yourself confused in whether to use a stemmer or a lemmatizer in your application. The following points might help you make the decision:

1. A stemmer is a rule based technique, and hence, it is much faster than the lemmatizer (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 lemmatizer.
2. A lemmatizer is slower because of the dictionary lookup but gives better results than a stemmer. Now, as a side note, it is important to know that for a lemmatizer to perform accurately, you need to provide the **part-of-speech tag** of the input word (noun, verb, adjective etc.). You’ll see learn POS tagging later - but it would suffice to know that there are often cases when the POS tagger itself is quite inaccurate on your text, and that will worsen the performance of the lemmatiser as well. In short, you may want to consider a stemmer rather than a lemmatiser if you notice that POS tagging is inaccurate.

**TF-IDF Representation**

The bag of words representation, while effective, is a very naive way of representing text. It relies on just the word frequencies of the words of a document. But don’t you think word representation shouldn’t solely rely on the word frequency? There is another way to represent documents in a matrix format which represents a word in a smarter way. It’s called the TF-IDF representation and it is the one that is often preferred by most data scientists.

The term TF stands for term frequency, and the term IDF stands for inverse document frequency. How is this different from bag-of-words representation?

The TF-IDF representation also called the **TF-IDF model**, considers the importance of each word. In the bag-of-words model, each word is assumed to be equally important, which is of course not correct.

The formula to calculate the TF-IDF weight of a term in a document is:

The log in the above formula is with base 10. Now, the tf-idf score for any term in a document is just the product of these two terms:

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

Note that tf-idf is implemented in different ways in different languages and packages. In the **tf score**representation, some people use only the frequency of the term, i.e. they don’t divide the frequency of the term with the total number of terms. In the **idf score**representation, some people use natural log instead of the log with base 10. Due to this, you may see a different score of the same terms in the same set of documents. But the goal remains the same - assign a weight according to the word's importance.

**Canonicalization**

It turns out that the stemming and lemmatization techniques are a part of what is known as canonicalization. Simply put, canonicalization means to reduce a word to its base form. Stemming and lemmatization were just specific instances of it. Stemming tries to reduce a word to its root form. Lemmatization tries to reduce a word to its lemma. The root and the lemma are nothing but the base forms of the inflected words.

Some cases can’t be handled either by stemming nor lemmatization. You need another preprocessing method in order to stem or lemmatize the words efficiently.

Suppose, you are working on a text corpus which contains misspelt words. Suppose, the corpus contains two misspelt versions of the word ‘disappearing’ - ‘dissappearng’  and ’disapearing’. After you stem these words, you’ll have two different stems - ‘dissappear’ and ‘disapear’. You still have the problem of redundant tokens. On the other hand, lemmatization won’t even work on these two words and will return the same words if it is applied because it only works on correct dictionary spelling.

To deal with misspellings, you’ll need to canonicalise it by correcting the spelling of the word. Then you can perform either stemming or lemmatization. The concept of **edit distance** which can then be used to build a spell corrector to rectify the spelling errors in the text that you’re working with.

A similar problem is that of pronunciation which has to do with different dialects present in the same language. For example, the word ‘colour’ is used in British English, while ‘color’ is used in American English. Both are correct spellings, but they have the exact same problem -  ‘colouring’ and ‘coloring’ will result in different stems and lemma.

To deal with different spellings that occur due to different pronunciations, the concept of **phonetic hashing** which will help you canonicalise different versions of the same word to a base word.

**Phonetic Hashing**

There are certain words which have different pronunciations in different languages. As a result, they end up being spelt differently. Examples of such words include names of people, city names, names of dishes, etc. Take, for example, the capital of India - New Delhi. Delhi is also pronounced as Dilli in Hindi. Hence, it is not surprising to find both variants in an uncleaned text corpus. Similarly, the surname ‘Agrawal’ has various spellings and pronunciations. Performing stemming or lemmatization to these words will not help us as much because the problem of redundant tokens will still be present. Hence, we need to reduce all the variations of a particular word to a common word.

To achieve this, you’ll need to know about what is called as the **phonetic hashing** technique.

Phonetic hashing buckets all the similar phonemes (words with similar sound or pronunciation) into a single bucket and gives all these variations a single hash code. Hence, the word ‘Dilli’ and ‘Delhi’ will have the same code.

Phonetic hashing is done using the Soundex algorithm. American Soundex is the most popular Soundex algorithm. It buckets British and American spellings of a word to a common code. It doesn't matter which language the input word comes from - as long as the words sound similar, they will get the same hash code.

Now, let’s arrive at the Soundex of the word ‘Mississippi’. To calculate the hash code, you’ll make changes to the same word, in-place, as follows:

1. Phonetic hashing is a four-letter code. The first letter of the code is the first letter of the input word. Hence it is retained as is. The first character of the phonetic hash is ‘M’. Now, we need to make changes to the rest of the letters of the word.
2. Now, we need to map all the consonant letters (except the first letter). All the vowels are written as is and ‘H’s, ‘Y’s and ‘W’s remain unencoded (unencoded means they are removed from the word). After mapping the consonants, the code becomes MI22I22I11I.

A close-up of a number

Description automatically generated

1. The third step is to remove all the vowels. ‘I’ is the only vowel. After removing all the ‘I’s, we get the code M222211. Now, you would need to merge all the consecutive duplicate numbers into a single unique number. All the ‘2’s are merged into a single ‘2’. Similarly, all the ‘1’s are merged into a single ‘1’. The code that we get is M21.
2. The fourth step is to force the code to make it a four-letter code. You need to pad it with zeroes in case it is less than four characters in length. Or you need to truncate it from the right side in case it is more than four characters in length. Since the code is less than four characters in length, you’ll pad it with one ‘0’ at the end. The final code is M210.

**Edit Distance**

Misspellings need to be corrected in order to stem or lemmatize efficiently. The problem of misspellings is so common these days, especially in text data from social media, that it makes working with text extremely difficult, if not dealt with.

Now, to handle misspellings, we need to make a **spell corrector**. All the misspelt words will be corrected to the correct spelling. In other words, all the misspelt words will be canonicalised to the base form, which is the correct spelling of that word. But to really understand how a spell corrector works, you’ll need to understand the concept of **edit distance**.

An edit distance is a distance between two strings which is a non-negative integer number.

an edit distance is the number of edits that are needed to convert a source string to a target string.

Now, the question that comes to the mind is - what’s an edit? An edit operation can be one of the following:

1. **Insertion** of a letter in the source string. To convert ‘color’ to ‘colour’, you need to insert the letter ‘u’ in the source string.
2. **Deletion** of a letter from the source string. To convert ‘Matt’ to ‘Mat’, you need to delete one of the ‘t’s from the source string.
3. **Substitution** of a letter in the source string. To convert ‘Iran’ to ‘Iraq’, you need to substitute ‘n’ with ‘q’

Now, it is easy to tell the edit distance between two relatively small strings. You can probably tell the number of edits that are needed in the string ‘applaud’ to ‘apple’. Did you guess how many? You need three edits. Substitution of ‘a’ to ‘e’ in a single edit. Then you require two deletions - deletion of the letters ‘u’ and ‘d’. Hence, you need a total of three edit operations in this case. But, this was a fairly simple example. It would become difficult when the two strings are relatively large and complex. Try calculating the edit distance between ‘deleterious’ and ‘deletion’. It’s not obvious in the first look. Hence, we need to learn how to calculate edit distance between any two given strings, however long and complex they might be.

A grid of numbers and letters

Description automatically generated

So, that’s how the Levenshtein edit distance is calculated.

Another variation of the edit distance - the Damerau–Levenshtein distance. The Damerau–Levenshtein distance, apart from allowing the three edit operations, also allows the swap (transposition) operation between two adjacent characters which costs only one edit instead of two.

This edit operation was introduced because swapping is a very common mistake. For example, while typing, people mistype ‘relief’ to ‘releif’. This has to be accounted as a single mistake (one edit distance), not two.

**Pointwise Mutual Information**

There is another common scenario that you’ll encounter while working with text. Suppose there is an article titled “Higher Technical Education in India” which talks about the state of Indian education system in engineering space. Let’s say, it contains names of various Indian colleges such as ‘International Institute of Information Technology, Bangalore’, ‘Indian Institute of Technology, Mumbai’, ‘National Institute of Technology, Kurukshetra’ and many other colleges. Now, when you tokenise this document, all these college names will be broken into individual words such as ‘Indian’, ‘Institute’, ‘International’, ‘National’, ‘Technology’ and so on. But you don’t want this. You want an entire college name to be represented by one token.

To solve this issue, you could either replace these college names by a single term. So, ‘International Institute of Information Technology, Bangalore’ could be replaced by ‘IIITB’. But this seems like a really manual process. To replace words in such manner, you would need to read the entire corpus and look for such terms.

Turns out that there is a metric called the **pointwise mutual information**, also called the **PMI**. You can calculate the PMI score of each of these terms. PMI score of terms such as ‘International Institute of Information Technology, Bangalore’ will be much higher than other terms. If the PMI score is more than a certain threshold then you can choose to replace these terms with a single term such as ‘International\_Institute\_of\_Information\_Technology\_Bangalore’.

To calculate PMI of a term that has two words we calculate PMI score:

PMI(x, y) = log ( P(x, y)/P(x)P(y) )

For terms with three words, the formula becomes:

PMI(z, y, x) = log [(P(z,y,x))/(P(z)P(y)P(x))]

= log [(P(z|y, x)\*P(y|x))\*P(x)/(P(z)P(y)P(x))]

= log [(P(z|y, x)\*P(y|x))/([P(z)P(y))]

Example: calculation of PMI(New Delhi) should be**log ( P(New Delhi)/P(New)P(Delhi) )**

To calculate the probability of your word you chose words as the occurrence context. But you could also choose a sentence or even a paragraph as the occurrence context.

If we choose **words** **as the occurrence context**, then the probability of a word is:

P(w) = Number of times given word ‘w’ appears in the text corpus/ Total number of words in the corpus

Similarly, if a **sentence** **is the occurrence context**, then the probability of a word is given by:

P(w) = Number of sentences that contain ‘w’ / Total number of sentences in the corpus

Similarly, you could calculate the probability of a word with paragraphs as occurrence context.

Once you have the probabilities, you can simply plug in the values and have the PMI score.

Now, calculating PMI score for a two-word term was pretty straightforward. But when you try to calculate the PMI of a three-word term such as “Indian Institute of Technology”, you will have to calculate P(Indian Institute Technology). To calculate such probability, you need to apply the chain rule of probability.

In practical settings, calculating PMI for terms whose length is more than two is still very costly for any relatively large corpus of text. You can either go for calculating it only for a two-word term or choose to skip it if you know that there are only a few occurrences of such terms.

You can also refer to [Palmetto](https://palmetto.demos.dice-research.org/)tool (In case this link does not work, please refer to [this](https://en.cs.uni-paderborn.de/ds/news-single/palmetto-a-quality-measuring-tool-for-topics)blog for more information about this tool) for calculating PMIs.

After calculating the PMI score, you can compare it with a cutoff value and see if PMI is larger or smaller than the cutoff value. A good cutoff value is zero. Terms with PMI larger than zero are valid terms, i.e. they don’t need to be tokenised into different words. You can replace these terms with a single-word term that has an underscore present between different words of the term. For example, the term ‘New Delhi’ has a PMI greater than zero. It can be replaced with ‘New\_Delhi’. This way, it won’t be tokenised while using the NLTK tokeniser.