M01S03-Advanced Lexical Processing

Natural Language Processing - Lexical Processing

Introduction

Basic lexical processing techniques are removing stop words, tokenisation, stemming and lemmatization followed by creating bag-of-words and tf-idf models and finally building a spam detector. These pre-processing steps are applicable in almost every text analytics application.

Even after going through all those pre-processing steps, a lot of noise is still present in the data. For example, **spelling mistakes** which happen by mistake as well as by choice (informal words such as 'lol', 'awsum' etc.).

1. How to identify and process incorrectly spelt words?
2. How to deal with spelling variations of a word that occur due to different pronunciations (e.g. Bangalore, Bengaluru)?
3. How to tokenise text efficiently?

One problem with the simple tokenisation approach is that it can't detect terms made up of more than one word. Terms such as 'Hong Kong', 'Calvin Klein', 'International Institute of Information Technology', etc. are made of more than one word, whereas they represent the same 'token'. There is no reason why we should have 'Hong' and 'Kong' as separate tokens. Techniques for building such intelligent tokenizers.

In this session:

- Phonetic hashing and the Soundex algorithm to handle different pronunciations of a word
- The minimum-edit-distance algorithm and building a spell corrector
- Pointwise mutual information (PMI) score to preserve terms that comprise of more than one word

Prerequisites

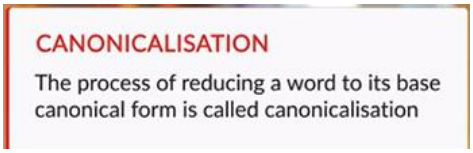
There are no prerequisites for this session, other than, knowledge of the previous session and the previous module.

Canonicalisation

Some techniques to help reduce a word to its base form.

1. Stemming
2. Lemmatization

Above techniques are **canonicalization** tools. Canonicalisation 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.



There are some cases that can't be handled either by stemming nor lemmatization. Another pre-processing method to stem or lemmatize the words efficiently.

Suppose, a text corpus with misspelt words contains two misspelt versions of the word 'disappearing' - 'dissappearng' and 'disappearing'. After stemming these words, two different stems - 'dissappear' and 'dissapear' which are redundant tokens. 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, canonicalise it by correcting the spelling of the word. Then perform either stemming or lemmatization. Concept of **edit distance** can be used to build a spell corrector to rectify the spelling errors in the corpus.

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.

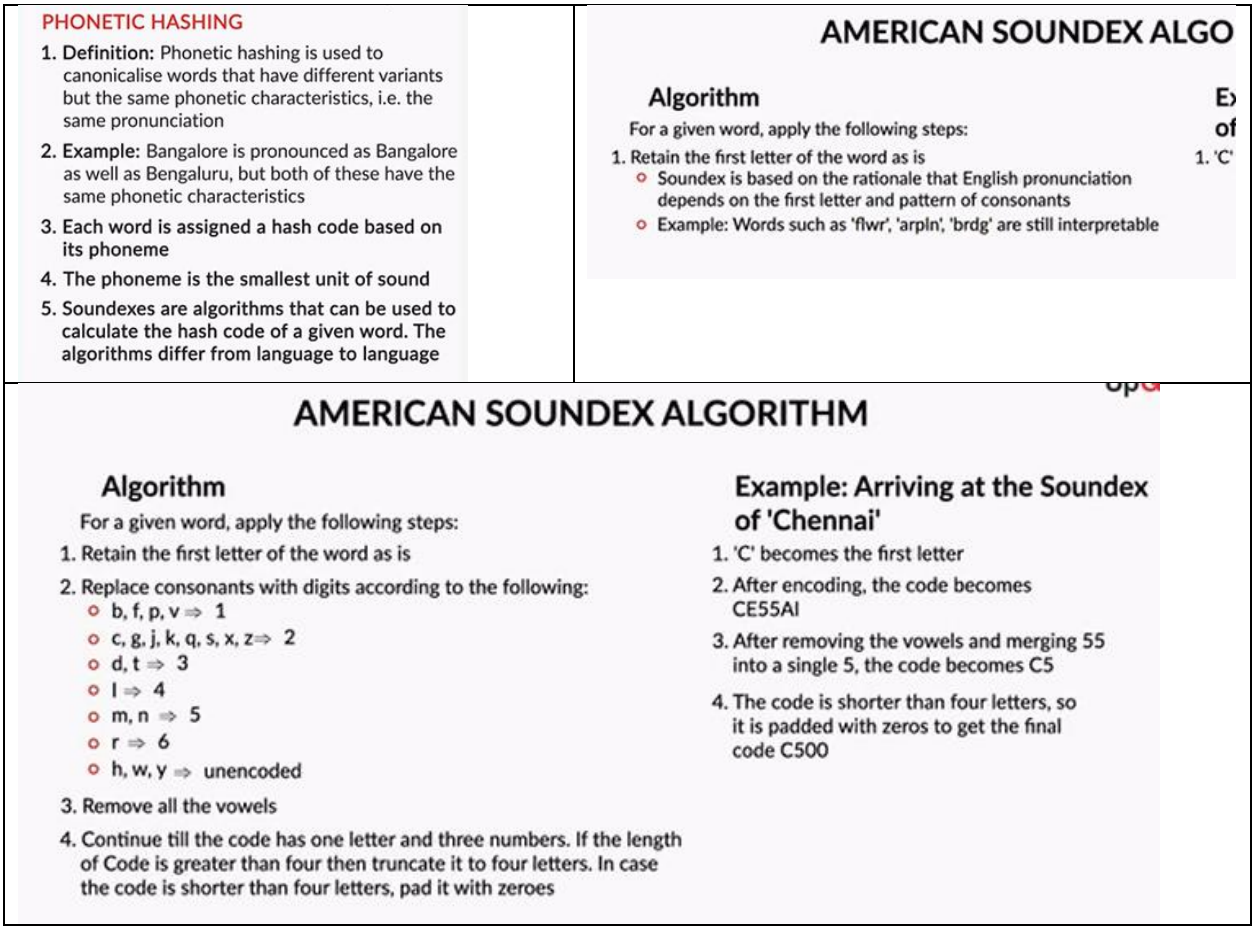
To deal with different spellings that occur due to different pronunciations, **phonetic hashing** is used to canonicalise different versions of the same word to a base word. Next section will cover phonetic hashing and how to use it to canonicalise words that have different spellings due to different pronunciations.

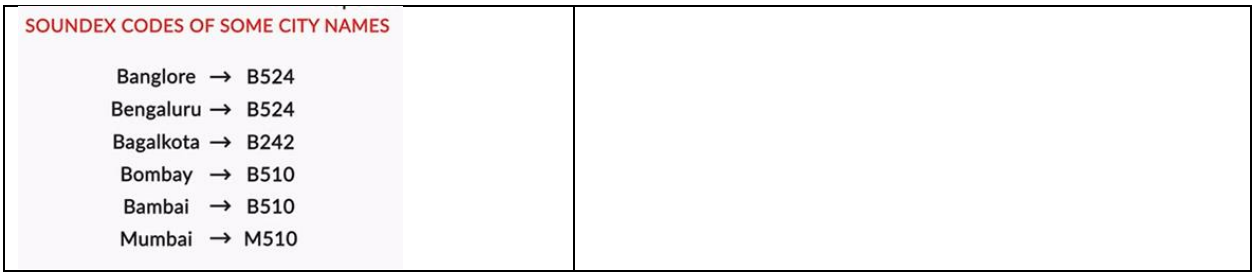
Phonetic Hashing

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 word to a common word.

To achieve this, we use **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.

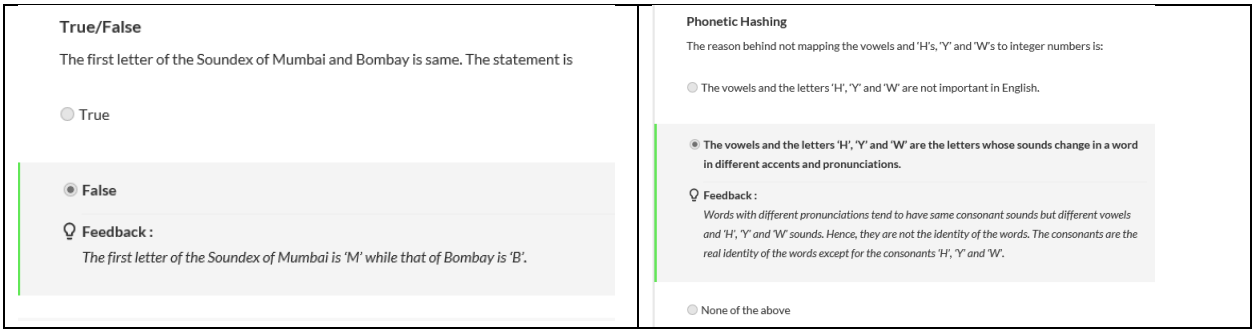


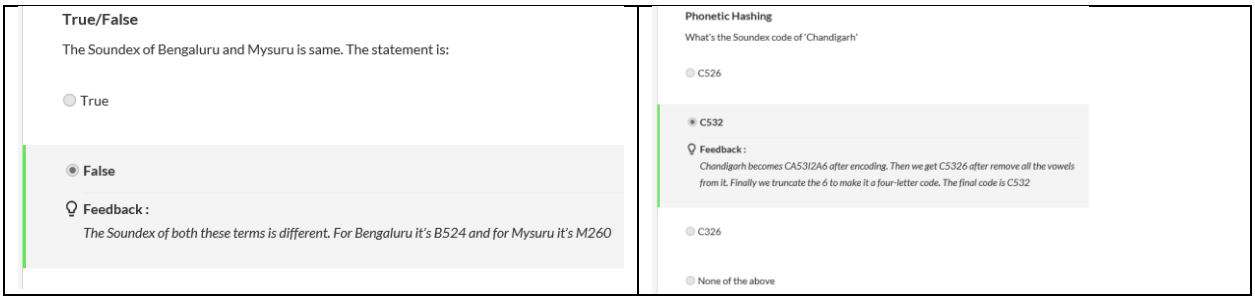


Phonetic hashing is done using the Soundex algorithm. American Soundex is the most popular Soundex algorithm. It buckets British and American spellings of a word to a common code. It doesn't matter which language the input word comes from - if the words sound similar, they will get the same hash code.

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.
3. 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 is merged into a single '2'. Similarly, all the '1's are merged into a single '1'. The code that we get is M21.
4. The fourth step is to force the code to make it a four-letter code. You either 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.





Since the process is fixed, we can simply create a function to create a Soundex code of any given input word.

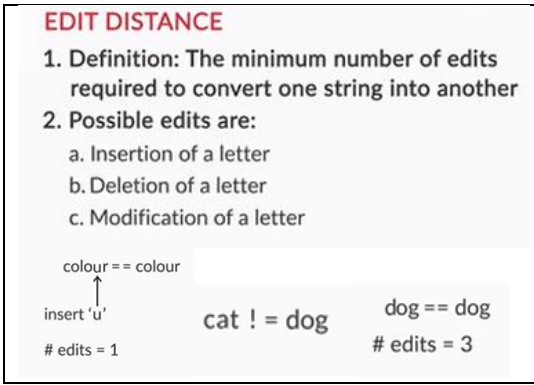
```
def get_soundex(token):
    """Get the soundex code for the string"""
    token = token.upper()

    soundex = ""

    # First letter of input is always the first letter of soundex
    soundex += token[0]

    # Create a dictionary which maps letters to respective soundex codes.
    # Vowels and 'H', 'W' and 'Y' will be represented by '.'
    dictionary = {"BFPV": "1", "CGJKQSXZ":"2", "DT":"3", "L":"4", "MN":"5", "R":"6", "AEIOUHWY":"."}

    for char in token[1:]:
        for key in dictionary.keys():
            if char in key:
                code = dictionary[key]
                if code != soundex[-1]:
                    soundex += code

    # Remove vowels and 'H', 'W' and 'Y' from soundex
    soundex = soundex.replace(".", "")

    # Trim or pad to make soundex a 4-character code
    soundex = soundex[:4].ljust(4, "0")

    return soundex
```

Edit Distance

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

To handle misspellings, how to make a **spell corrector** to work on all the misspelt words to get correct spelling? All misspelt words to be canonicalised to the base form, correct spelling of that word. To really understand how a spell corrector works, understand **edit distance**.

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



COMMONLY MISSPELLED WORDS

(in alphabetical order)

Correct spelling	Common misspellings
absence	absense
acceptable	acceptible
accidentally	accidentaly
achieve	acheive
accommodate	accomodate, acomodate
acquire	aquire, adquire
acquit	aquit
acreage	acrage, acerage
address	adress
adultery	adultary

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

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', insert the letter 'u' in the source string.
2. **Deletion** of a letter from the source string. To convert 'Matt' to 'Mat', delete one of the 't's from the source string.
3. **Substitution** of a letter in the source string. To convert 'Iran' to 'Iraq', substitute 'n' with 'q'

It is easy to tell the edit distance between two relatively small strings. It is easy to tell edit distance from 'applaud' to 'apple' which is 3, substitute 'a' with 'e' and delete 'u' and 'd'. It is fairly simple example. It would be difficult when two strings are relatively large and complex. Calculating the edit distance between 'deleterious' and 'deletion' is not obvious in the first look. We need to learn how to calculate edit distance between any two given strings, however long and complex they might be.

More importantly, we need an algorithm to compute the edit distance between two words.

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.								
a								
q								
u								
i								
r								
e								

We need distance from acquire to aquire

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.								
a								
q								
u								
i								
r								
e								

(M+1) x (N+1) = 7x8

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.								
a								
q								
u								
i								
r								
e								

. is null value

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.	0	1	2	3	4	5	6	7
a	1							
q	2							
u	3							
i	4							
r	5							
e	6							

Distance from null value to character

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.	0	1	2	3	4	5	6	7
a	1	0						
q	2							
u	3							
i	4							
r	5							
e	6							

Both are same so take the diagonal value

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.	0	1	2	3	4	5	6	7
a	1	0	0+1					
q	2							
u	3							
i	4							
r	5							
e	6							

Different letters so take 3 preceding neighbours and add 1 to min of them

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.	0	1	2	3	4	5	6	7
a	1	0	1					
q	2							
u	3							
i	4							
r	5							
e	6							

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.	0	1	2	3	4	5	6	7
a	1	0	1					
q	2	1						
u	3							
i	4							
r	5							
e	6							

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.	0	1	2	3	4	5	6	7
a	1	0	1	2	3	4	5	6
q	2	1	1	1	2	3	4	5
u	3	2	2	2	1	2	3	4
i	4	3	3	3	2	1	2	3
r	5	4	4	4	3	2	1	2
e	6	5	5	5	4	3	2	1

One edit operation from aq to a

Total edit distance is 1 on the bottom right cell.

This is how Levenshtein edit distance is calculated

Edit Distance

Suppose you are calculating the edit distance between the strings 'courageous' (the correct spelling) and 'courageus' (misspelt word). What is the value in the cell present in the row with letter 'g' and the column with letter 'g'?

You answered : 0 Correct Answer : 0

Feedback :

Correct. The value in the cell present in the row with letter 'g' and the column with the letter 'g is same as the edit distance between the words 'courag' and 'courag' which is zero.

	.	c	o	u	r	a	g	e	o	u	s
.	0	1	2	3	4	5	6	7	8	9	10
c	1	0									
o	2		0								
u	3			0							
r	4				0						
a	5					0					
g	6						0				
e	7										
u	8										

s 9

Edit Distance

Suppose you are calculating the edit distance between the strings 'courageous' (the correct spelling) and 'courageus' (misspelt word). What is the value in the cell present in the row with letter 'a' and the column with letter 'e'?

You answered : 2 Correct Answer : 2

Feedback :
Correct. The value in the cell present in the row with letter 'e' and the column with the letter 'a' is same as the edit distance between the words 'coura' and 'courage' which is two.

Edit Distance

What is the edit distance between 'Mountain' and 'Mountbatten'?

You answered : 4 Correct Answer : 4

Feedback :
Correct.

Mountain
Mountbatten

Mount, a and n are unchanged
i would be gone – 1
b would be added - 1
tte would be added – 3

Edit Distance

"The number of edits to transform source string to target string is the same as the number of edits to transform the target string to source string."

The above statement is true/false?

True **Correct**

Feedback :
The statement is correct. It doesn't matter which is the source string and which is the target string while calculating the edit distance.

False

Since the process of calculating edit distance is fixed, we can write an algorithm to automate this computation.

Levenshtein Edit Distance

The levenshtein distance calculates the number of steps (insertions, deletions or substitutions) required to go from source string to target string.

```
def lev_distance(source='', target=''):
    """Make a Levenshtein Distances Matrix"""

    # get length of both strings
    n1, n2 = len(source), len(target)

    # create matrix using length of both strings - source string sits on columns, target string sits on rows
    matrix = [ [ 0 for i1 in range(n1 + 1) ] for i2 in range(n2 + 1) ]

    # fill the first row - (0 to n1-1)
    for i1 in range(1, n1 + 1):
        matrix[0][i1] = i1

    # fill the first column - (0 to n2-1)
    for i2 in range(1, n2 + 1):
        matrix[i2][0] = i2

    # fill the matrix
    for i2 in range(1, n2 + 1):
        for i1 in range(1, n1 + 1):

            # check whether letters being compared are same
            if (source[i1-1] == target[i2-1]):
                value = matrix[i2-1][i1-1]               # top-left cell value
            else:
                value = min(matrix[i2-1][i1]   + 1,      # left cell value     + 1
                            matrix[i2][i1-1]   + 1,      # top cell  value     + 1
                            matrix[i2-1][i1-1] + 1)      # top-left cell value + 1

            matrix[i2][i1] = value

    # return bottom-right cell value
    return matrix[-1][-1]
```

Levenshtein distance in nltk library

```
# import library
from nltk.metrics.distance import edit_distance
```

```
edit_distance("apple", "appel")
```

```
2
```

Damerau-Levenshtein Distance

The Damerau-Levenshtein distance allows transpositions (swap of two letters which are adjacent to each other) as well.

```
: edit_distance("apple", "appel", transpositions=False, )
```

```
: 2
```

So that's how to compute the edit distance between two given strings. Another variation of the edit distance is - the Damerau–Levenshtein distance. This variation, 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 must be accounted as a single mistake (one edit distance), not two.

But how to make a spell corrector which was the main objective in the first place?

Spell Corrector - I

A spell corrector is a widely used application that you would see almost everywhere on the internet. If you have the autocorrect feature enabled on your phone, the incorrect spellings get replaced by the correct ones. Another example is when you use a search engine such as Google to search anything and mistype a word, it suggests the correct word.

Spell correction is an important part of lexical processing. In many applications, spell correction forms an initial preprocessing layer. For example, if you are making a chatbot to book flights, and you get the user request 'Book a flight from Mumbai to *Bangalor',* you want to gracefully handle that spelling error and return relevant results.

Now, people have made various attempts to make spell correctors using different techniques. Some are very basic and elementary which use lexical processing, while others are state-of-the-art performers which use deep learning architectures.

Here, you're going to learn the Norvig's spell corrector which gives you really good performance and result, given its simplicity.

Now, let's look at what each function does. The function words() is pretty straightforward. It tokenises any document that's passed to it. You have already learnt how to tokenise words using NLTK library. You could also use regular expressions to tokenise words. The 'Counter' class, which you just saw in the Jupyter notebook, creates a frequency distribution of the words present in the seed document. Each word is stored along with its count in a Python dictionary format. You could also use the NLTK's FreqDist() function to achieve the same results.

Seed document 'big.txt' is nothing but the book 'The Adventures of Sherlock Holmes' present in text format at project Gutenberg's website. A seed document acts like a lookup dictionary for a spell corrector since it contains the correct spellings of each word.

Now, you might ask why not just use a dictionary instead of a book? You'll get to know why we're using a book a little later.

The edits_one() function creates all the possible words that are one edit distance away from the input word. Most of the words that this function creates are garbage, i.e. they are not valid English words. For example, if you pass the word 'laern' (misspelling of the word 'learn') to edits_one(), it will create a list where the word 'lgern' will be present since it is an edit away from the word 'laern'. But it's not an English word. Only a subset of the words will be actual English words.

Similarly, the edits_two() function creates a list of all the possible words that are two edits away from the input word. Most of these words will also be garbage.

The **known()** function filters out the valid English word from a list of given words. It uses the frequency distribution as a dictionary that was created using the seed document. If the words created using edits_one() and edits_two() are not in the dictionary, they're discarded.

Now, the function **possible_corrections()** returns a list of all the potential words that can be the correct alternative spelling. For example, let's say the user has typed the word 'wut' which is wrong. There are multiple words that could be the correct spelling of this word such as 'cut', 'but', 'gut', etc. This function

will return all these words for the given incorrect word ‘wut’. But, how does this function find all these word suggestions exactly? It works as follows:

1. It first checks if the word is correct or not, i.e. if the word typed by the user is a present in the dictionary or not. If the word is present, it returns no spelling suggestions since it is already a correct dictionary word.
2. If the user types a word which is not a dictionary word, then it creates a list of all the **known** words that are **one edit** distance away. If there are no valid words in the list created by edits_one() only then this function fetches a list of all known words that are **two edits** away from the input word
3. If there are no known words that are two edits away, then the function returns the original input word. This means that there are no alternatives that the spell corrector could find. Hence, it simply returns the original word.

Finally, there is the **prob()** function. The function returns the probability of an input word. This is exactly why you need a seed document instead of a dictionary. A dictionary only contains a list of all correct English words. But, a seed document not only contains all the correct words but it could also be used to create a frequency distribution of all these words. This frequency will be taken into consideration when there are more than one possibly correct words for a given misspelled word. Let’s say the user has input the word ‘wut’. The correct alternative to this word could be one of these words - ‘cut’, ‘but’ and ‘gut’, etc. The possible_corrections() function will return all these words. But the prob() function will create a probably associated with each of these suggestions and return the one with highest probability. Suppose, if a word ‘but’ is present 2000 times out of a total of million words in the seed document, then it’s probability would be 2000/1000000, i.e. 0.002.

Spell Corrector - II

In this section, you'll continue to build the rest of the spell corrector. Till now, you've seen how to build the functions edits_one(), edits_two(), known(), possible_corrections() and prob(). Let's understand all these functions in more depth by taking a look at their outputs.

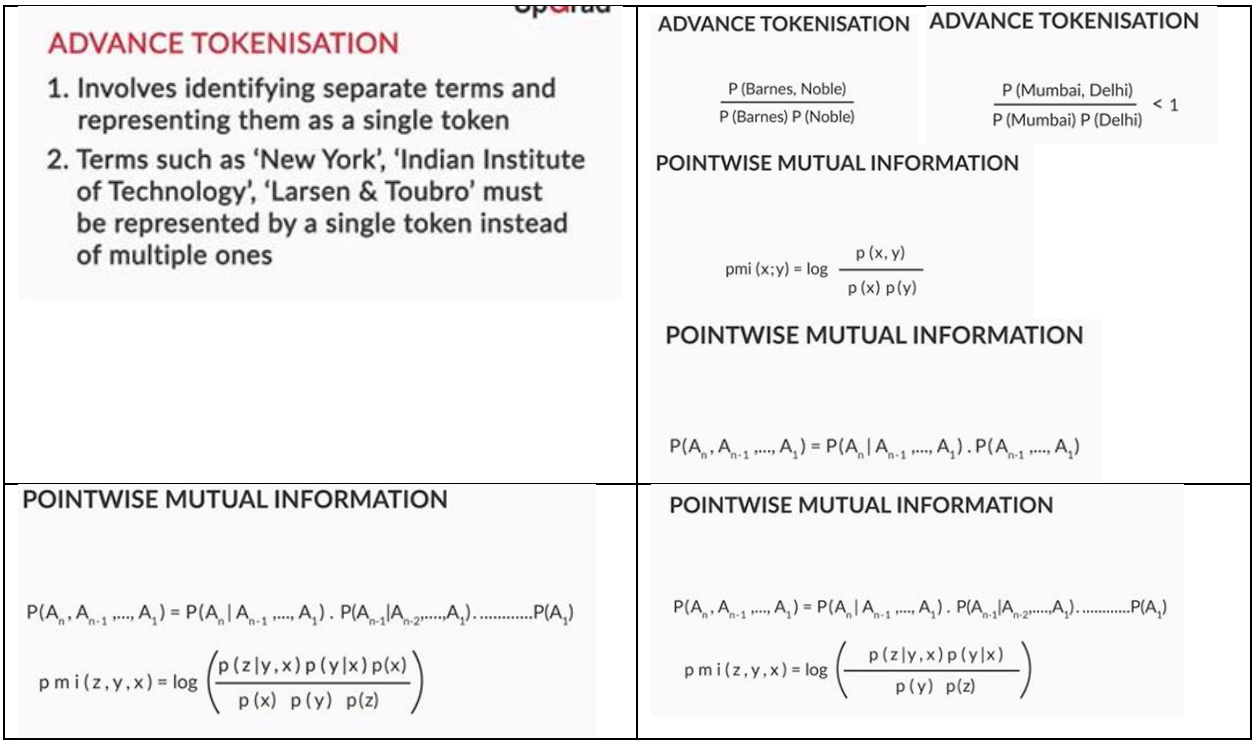
Pointwise Mutual Information - I

Till now you have learnt about reducing words to their base form. But 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 manual process. To replace words in such manner, you would need to read the entire corpus and look for such terms.

A metric **pointwise mutual information** or **PMI** calculates 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 than you can choose to replace these terms with a single term such as 'International_Institute_of_Information_Technology_Bangalore'.

But what is PMI and how is it calculated?



You saw how to calculate PMI of a term that has two words. The PMI score for such term is:

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

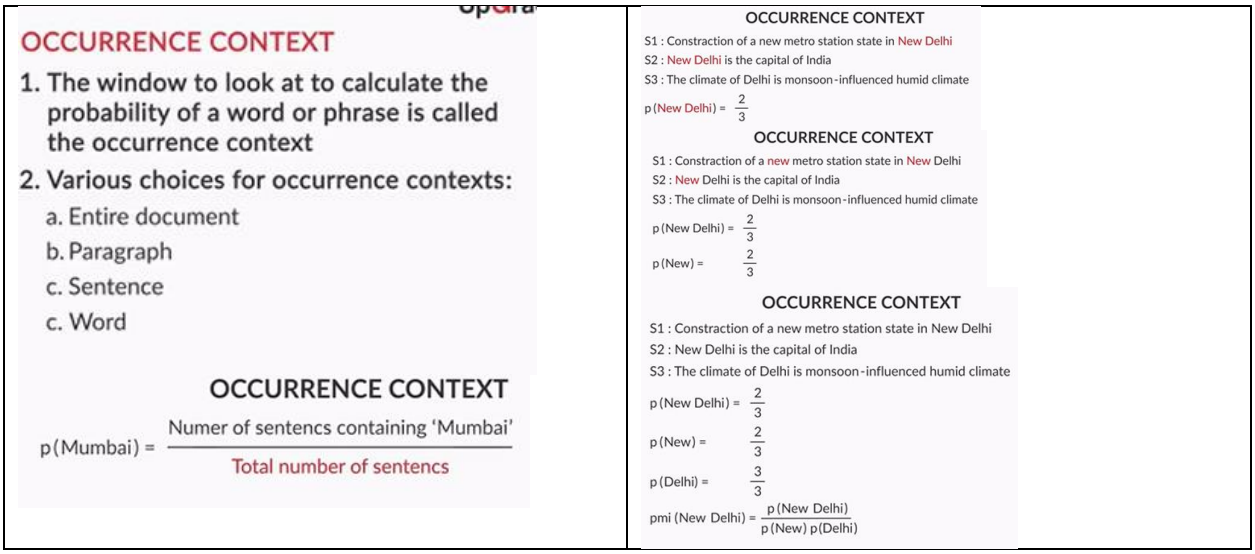
For terms with three words, the formula becomes:

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

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

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

Now, how do you calculate these probabilities?



Till now, 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.

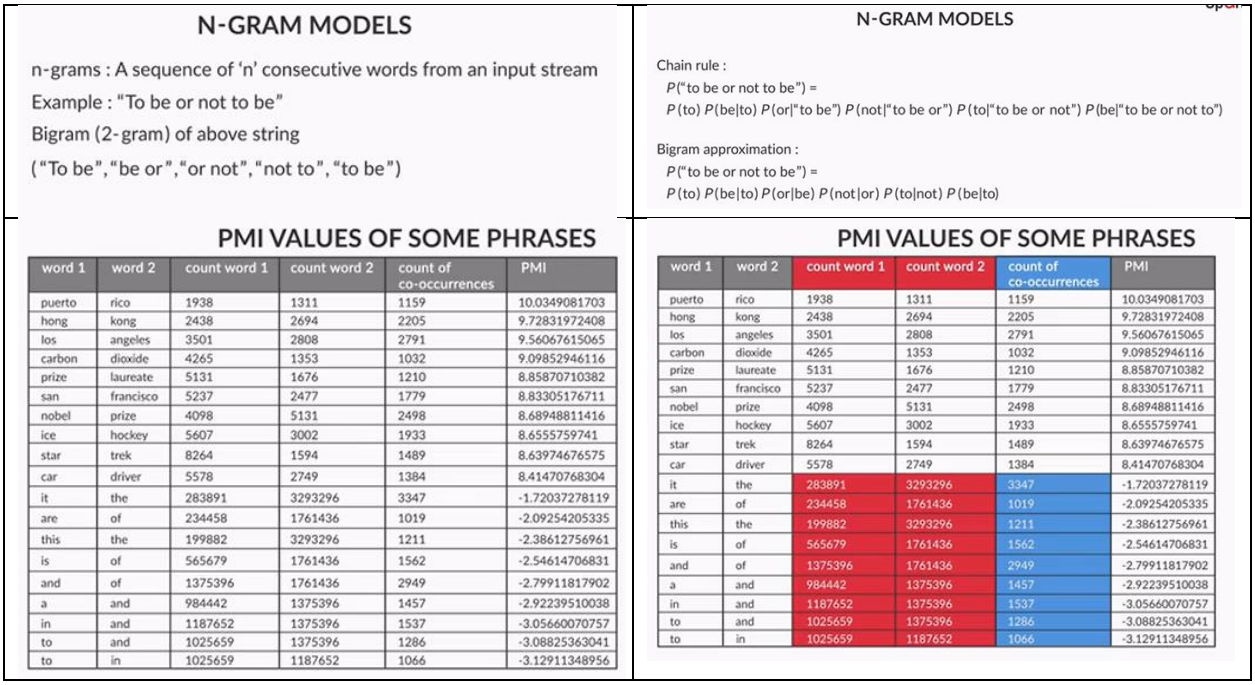
Now, you're given the following corpus of text:

"The Nobel Prize is a set of five annual international awards bestowed in several categories by Swedish and Norwegian institutions in recognition of academic, cultural, or scientific advances. In the 19th century, the Nobel family who were known for their innovations to the oil industry in Azerbaijan was the leading representative of foreign capital in Baku. The Nobel Prize was funded by personal fortune of Alfred Nobel. The Board of the Nobel Foundation decided that after this addition, it would allow no further new prize."

Consider the above corpus to answer the questions of the following exercise. Take each **sentence** of the corpus as the **occurrence context** and attempt the following exercise.

Pointwise Mutual Information - II

Now, calculating PMI score for a two-word term was 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.



N-GRAM MODELS

n-grams : A sequence of ‘n’ consecutive words from an input stream

Example : “To be or not to be”

Bigram (2-gram) of above string

(“To be”, “be or”, “or not”, “not to”, “to be”)

PMI VALUES OF SOME PHRASES

word 1	word 2	count word 1	count word 2	count of co-occurrences	PMI
puerto	rico	1938	1311	1159	10.0349081703
hong	kong	2438	2694	2205	9.72831972408
los	angeles	3501	2808	2791	9.56067615065
carbon	dioxide	4265	1353	1032	9.09852946116
prize	laureate	5131	1676	1210	8.85870710382
san	francisco	5237	2477	1779	8.83305176711
nobel	prize	4098	5131	2498	8.68948811416
ice	hockey	5607	3002	1933	8.6555759741
star	trek	8264	1594	1489	8.63974676575
car	driver	5578	2749	1384	8.41470768304
it	the	283891	3293296	3347	-1.72037278119
are	of	234458	1761436	1019	-2.09254205335
this	the	199882	3293296	1211	-2.38612756961
is	of	565679	1761436	1562	-2.54614706831
and	of	1375396	1761436	2949	-2.79911817902
a	and	984442	1375396	1457	-2.92239510038
in	and	1187652	1375396	1537	-3.05660070757
to	and	1025659	1375396	1286	-3.08825363041
to	in	1025659	1187652	1066	-3.12911348956

N-GRAM MODELS

Chain rule :

$P(\text{“to be or not to be”}) =$

$P(\text{to})\,P(\text{be}|\text{to})\,P(\text{or}|\text{“to be”})\,P(\text{not}|\text{“to be or”})\,P(\text{to}|\text{“to be or not”})\,P(\text{be}|\text{“to be or not to”})$

Bigram approximation :

$P(\text{“to be or not to be”}) =$

$P(\text{to})\,P(\text{be}|\text{to})\,P(\text{or}|\text{be})\,P(\text{not}|\text{or})\,P(\text{to}|\text{not})\,P(\text{be}|\text{to})$

PMI VALUES OF SOME PHRASES

word 1	word 2	count word 1	count word 2	count of co-occurrences	PMI
puerto	rico	1938	1311	1159	10.0349081703
hong	kong	2438	2694	2205	9.72831972408
los	angeles	3501	2808	2791	9.56067615065
carbon	dioxide	4265	1353	1032	9.09852946116
prize	laureate	5131	1676	1210	8.85870710382
san	francisco	5237	2477	1779	8.83305176711
nobel	prize	4098	5131	2498	8.68948811416
ice	hockey	5607	3002	1933	8.6555759741
star	trek	8264	1594	1489	8.63974676575
car	driver	5578	2749	1384	8.41470768304
it	the	283891	3293296	3347	-1.72037278119
are	of	234458	1761436	1019	-2.09254205335
this	the	199882	3293296	1211	-2.38612756961
is	of	565679	1761436	1562	-2.54614706831
and	of	1375396	1761436	2949	-2.79911817902
a	and	984442	1375396	1457	-2.92239510038
in	and	1187652	1375396	1537	-3.05660070757
to	and	1025659	1375396	1286	-3.08825363041
to	in	1025659	1187652	1066	-3.12911348956

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.

After calculating the PMI score, you can compare it with a cut off value and see if PMI is larger or smaller than the cut off value. A good cut off 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.

Summary

In this session, you learnt about how to deal with three scenarios:

1. Handling differently spelt words due to different pronunciations
2. Correcting spelling of misspelt words using edit distance
3. Tokenising terms that comprise of multiple words

To handle words that have different spellings due to different pronunciations, you learnt the concept of **phonetic hashing**. Phonetic hashing is used to bucket words with similar pronunciation to the same hash code. To hash words, you used the **Soundex** algorithm. The American Soundex algorithm maps the letters of a word in such a way that words are reduced to a four-character long code. Words with the same Soundex code can be replaced by a common spelling of the word. This way, you learnt how to get rid of different variations in spellings of a word.

The next thing that you learnt about was the **Levenshtein edit distance** and spell corrector. You learnt that an edit distance is the number of edits that are needed to convert a source string to a target string. In a single edit operation, you can either insert, delete or substitute a letter. You also learnt a different variant of edit distance - the **Damerau–Levenshtein distance**. It lets you swap two adjacent letters in a single edit operation.

With the help of the edit distance, you created a spell corrector. You could use that spell corrector to rectify the spelling of incorrect words in your corpus.

Lastly, you learnt about the **pointwise mutual information (PMI) score**. You saw how you can calculate PMI of terms with two or more words. You learnt about the concept of **occurrence context**. After choosing the occurrence context, you can calculate the PMI of a term and choose whether it is a valid term or not based on the cutoff value. A good cutoff value is zero. Terms that have PMI higher than zero can be replaced by a single term by simply attaching the multiple words using an underscore.