# Introduction

In the previous session, you had learnt all the basic lexical processing techniques such as removing stop words, tokenisation, stemming and lemmatization followed by creating bag-of-words and tf-idf models and finally building a spam detector. These preprocessing steps are applicable in almost every text analytics application.

Even after going through all those preprocessing steps that you learnt in the previous session, 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.). To handle such situations, you'll learn how to identify and process incorrectly spelt words. Also, you'll learn how to deal with spelling variations of a word that occur due to different pronunciations (e.g. Bangalore, Bengaluru).

At the end of the session, you'll also learn how to tokenise text efficiently. You've already learnt how to tokenise words, but one problem with the simple tokenisation approach is that it can't detect terms that are 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. You'll study techniques for building such intelligent tokenizers.

In this session, you'll learn:

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

## Canonicalisation

In the last session, you had learnt some techniques that help you reduce a word to its base form. Specifically, you had learnt the following techniques:

Stemming

Lemmatization

It turns out that the above techniques are a part of what is known as canonicalisation. Simply put, 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.

In the following lecture, professor Srinath explains the concept of canonicalisation.


## HANDLING REDUNDANT TOKENS

### Various forms of Agrawal

Agarwal, Aggarwal, Aggarwal, etc.

### British versus American spelling

| British spelling | American spelling |
|---|---|
| Colour | Color |
| Theatre | Theater |
| Defence | Defense |
| Travelling | Traveling |
| Tokenise | Tokenize |

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, you’ll learn the concept of phonetic hashing which will help you canonicalise different versions of the same word to a base word.

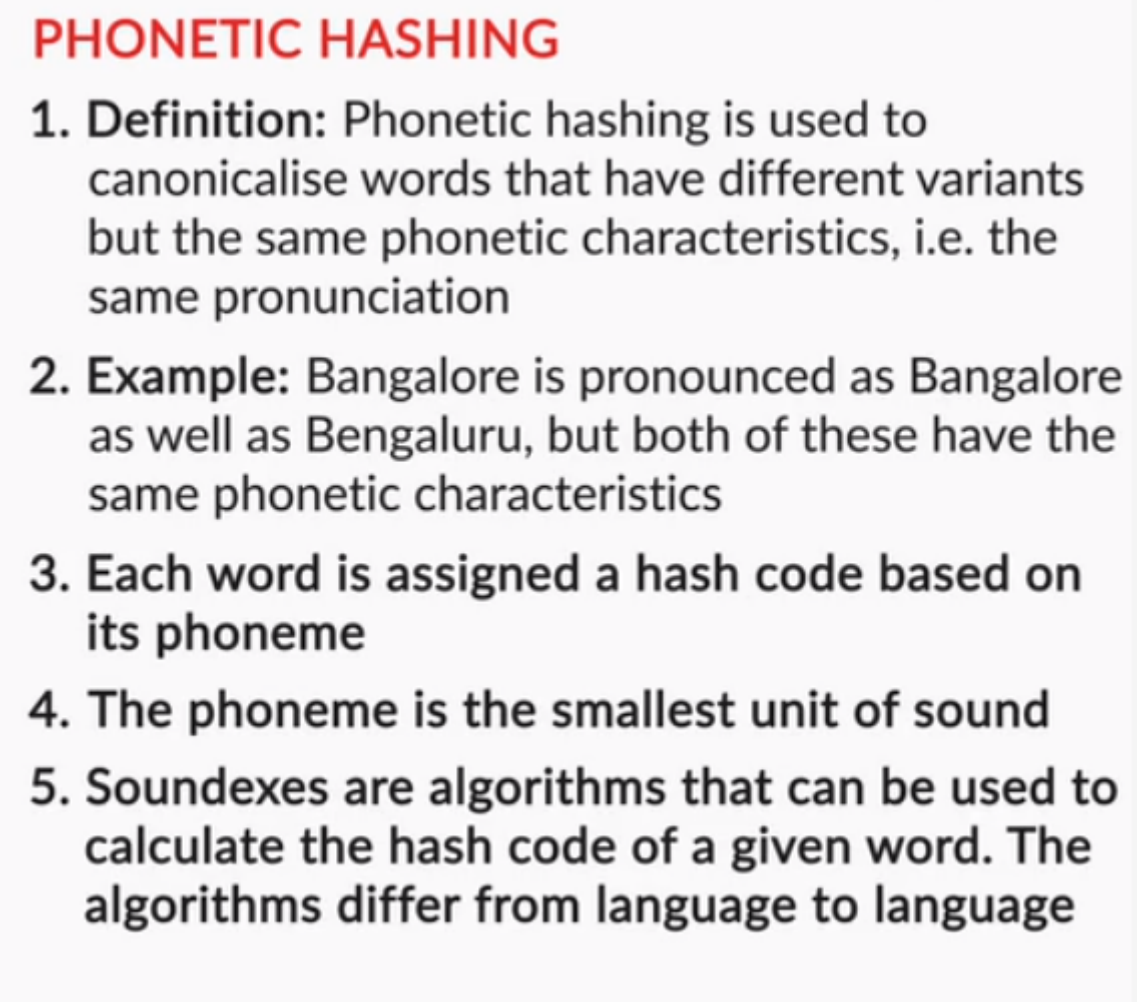
In the next section, you'll learn about phonetic hashing and how to use it to canonicalise words that have different spellings due to different pronunciations.

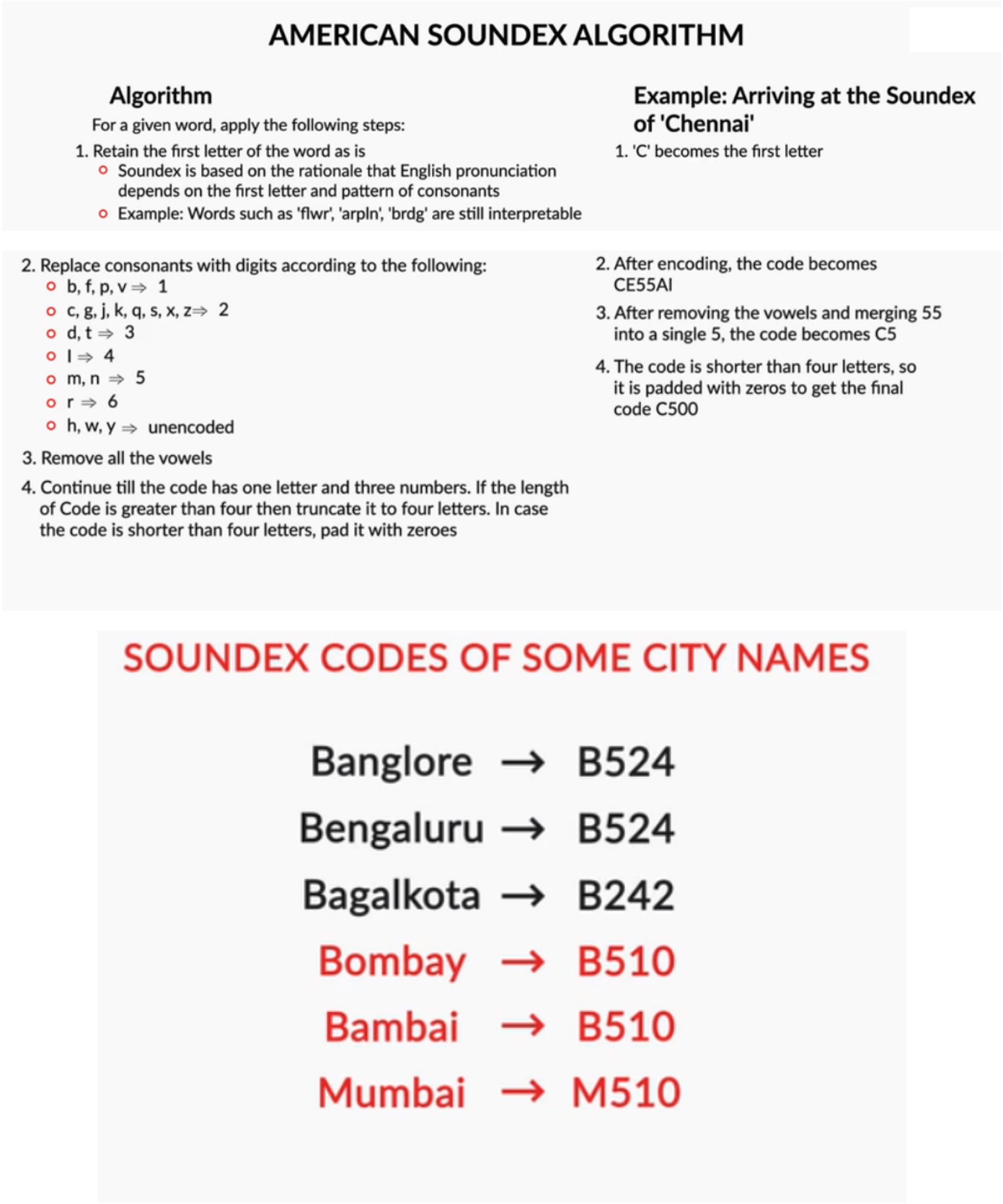
## Phonetic Hashing

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 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:

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.

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.

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.

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. Learn how to make such function from professor Srinath as he explains this with a Jupyter notebook. Download the Jupyter notebook from the link given below to follow along

# Soundex

***Let's create a function which calculates the soundex of any given string***

In [1]:

```
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', '
    dictionary = {"BFPV": "1", "CGJKQSXZ":"2", "DT":"3", "L":"4", "MN":"5", "R":"6", "AEIOU

    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
```

## Let's see what's the soudex of 'Bombay' and 'Bambai'

In [2]:

```
print(get_soundex("Bombay"))
print(get_soundex("Bambai"))
```

```
B510
B510
```

### Let's see soundex of 'Aggrawal', 'Agrawal', 'Aggarwal' and 'Agarwal'

In [3]:

```
print(get_soundex("Aggrawal"))
print(get_soundex("Agrawal"))
print(get_soundex("Aggarwal"))
print(get_soundex("Agarwal"))
```

```
A264
A264
A264
A264
```

Up next, you'll learn how to identify and measure the 'distance between words' using the concept of edit distance which will help you build your own spell corrector.

## Edit Distance

In the last section, you saw how to deal with different pronunciations of a particular word. Next, you'll learn how to deal with misspellings. As already discussed, 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, you'll learn how 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. Professor Srinath explains the concept of edit distance in the following lecture.

## EDIT DISTANCE

1. Definition: The minimum number of edits required to convert one string into another
2. Possible edits are:
   a. Insertion of a letter
   b. Deletion of a letter
   c. Modification of a letter

color != colour

colour = = colour
↑
insert 'u'

# edits = 1

cat ! = dog

dog == dog
# edits = 3

## 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 |

As you just learnt, 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:

Insertion of a letter in the source string. To convert ‘color’ to ‘colour’, you need to insert the letter ‘u’ in the source string.

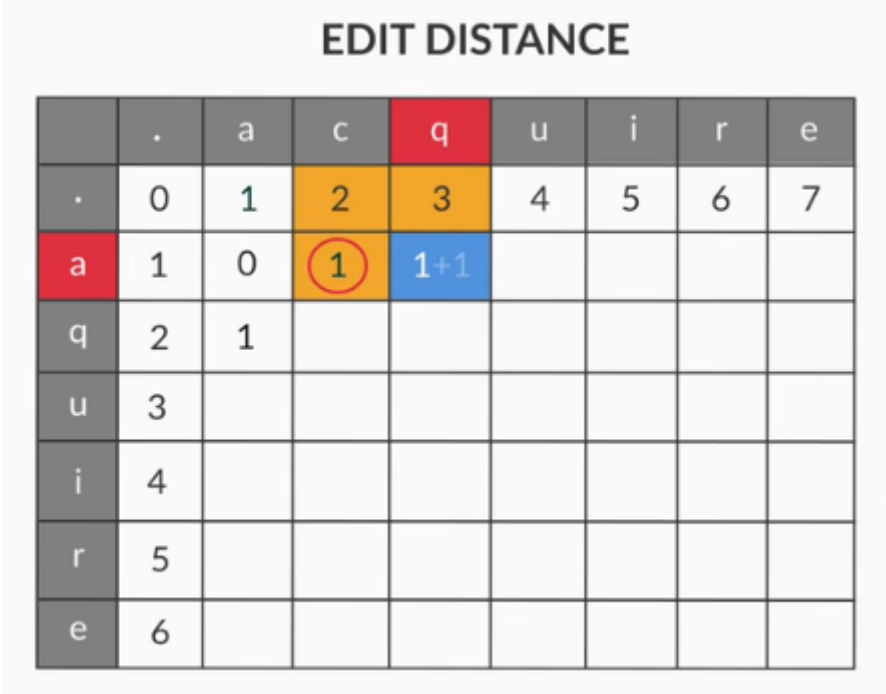
Deletion of a letter from the source string. To convert ‘Matt’ to ‘Mat’, you need to delete one of the ‘t’s from the source string.

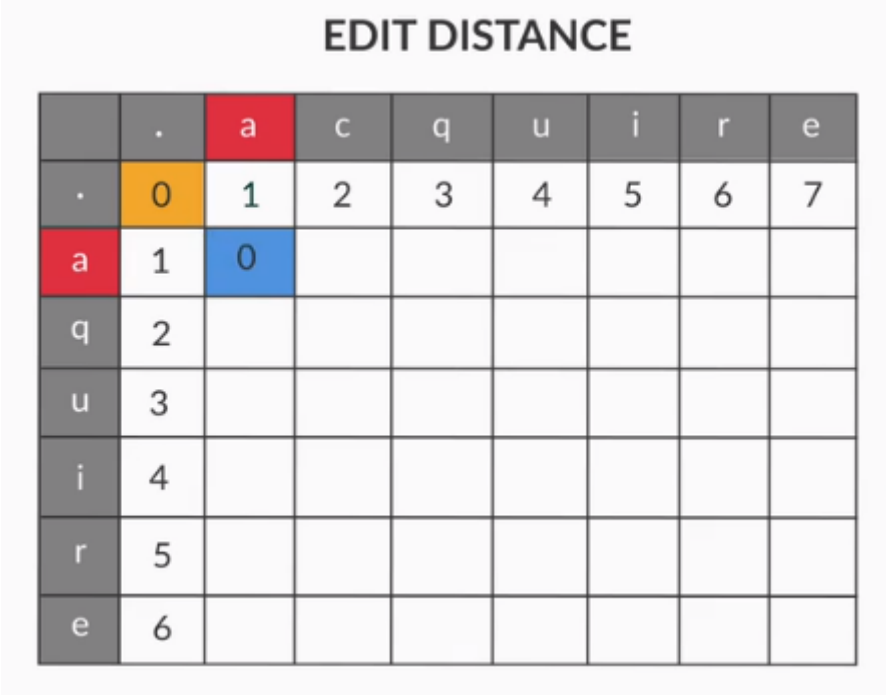
Substitution of a letter in the source string. To convert ‘Iran’ to ‘Iraq’, you need to substitute ‘n’ with ‘q’

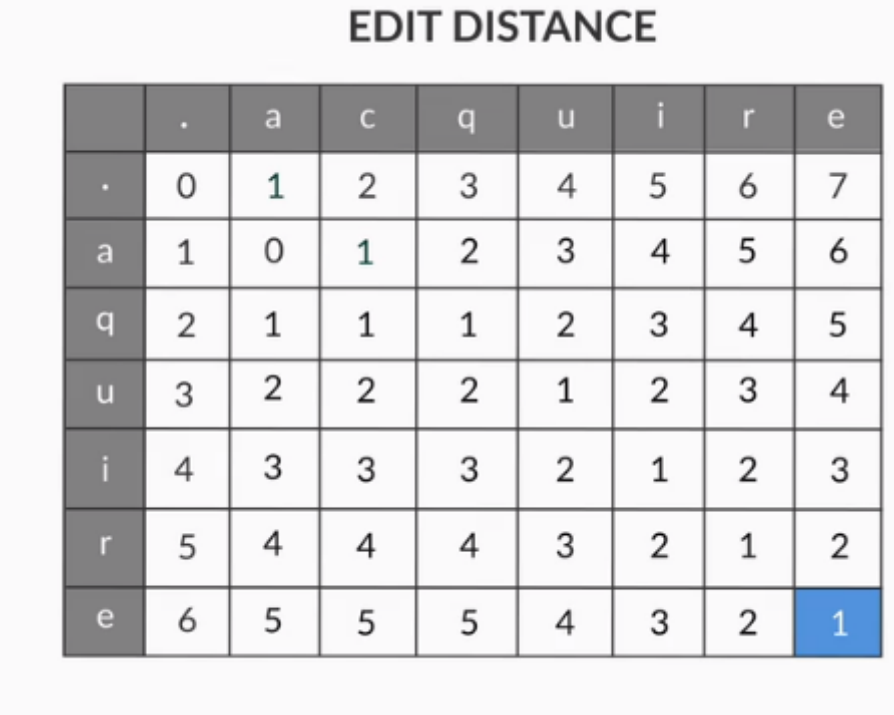
Now, it is easy to tell the edit distance between two relatively small strings. You can probably tell the number of edits that are needed in the string ‘applaud’ to ‘apple’. Did you guess how many? You need three edits. Substitution of ‘a’ to ‘e’ in a single edit. Then you require two deletions - deletion 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.

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.

More importantly, we need an algorithm to compute the edit distance between two words. Professor Srinath explains such an algorithm in the following lecture.






sp the Edit distance between acquire and aquire is " 1 "

So, that's how the Levenshtein edit distance is calculated. Now, attempt the following exercise to practice and strengthen the concept of edit distance.

## Levenshtein Edit Distance

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

In [4]:

```
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 st
    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]
```

In [5]:

```
lev_distance('Mountain', 'Mountbatten')
```

Out[5]:

```
4
```

## Levenshtein distance in nltk library

In [6]:

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

In [34]:

```
edit_distance("perspective", "prospective")
```

Out[34]:

2

### Damerau-Levenshtein Distance

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

In [8]:

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

Out[8]:

2

So that's how you compute the edit distance between two given strings. You also saw 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.

But how to make a spell corrector which was the main objective in the first place? You'll learn to do that in the next section.

## 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. Watch the following video where Krishna explains how to build a Norvig spell corrector.

In [10]:

```
import re
from collections import Counter

# function to tokenise words
def words(document):
    "Convert text to lower case and tokenise the document"
    return re.findall(r'\w+', document.lower())
```

In [11]:

```
# create a frequency table of all the words of the document
all_words = Counter(words(open('big.txt').read()))
```

In [12]:

```
# check frequency of a random word, say, 'chair'
all_words['chair']
```

Out[12]:

```
135
```

In [13]:

```
# look at top 10 frequent words
all_words.most_common(10)
```

Out[13]:

```
[('the', 79809),
 ('of', 40024),
 ('and', 38312),
 ('to', 28765),
 ('in', 22023),
 ('a', 21124),
 ('that', 12512),
 ('he', 12401),
 ('was', 11410),
 ('it', 10681)]
```

In [14]:

```
def edits_one(word):
    "Create all edits that are one edit away from `word`."
    alphabets    = 'abcdefghijklmnopqrstuvwxyz'
    splits     = [(word[:i], word[i:])                  for i in range(len(word) + 1)]
    deletes    = [left + right[1:]                      for left, right in splits if right
    inserts    = [left + c + right                      for left, right in splits for c in
    replaces   = [left + c + right[1:]                  for left, right in splits if right
    transposes = [left + right[1] + right[0] + right[2:] for left, right in splits if len(r
    return set(deletes + inserts + replaces + transposes)
```

In [15]:

```
def edits_two(word):
    "Create all edits that are two edits away from `word`."
    return (e2 for e1 in edits_one(word) for e2 in edits_one(e1))
```

In [16]:

```
edits_two('chair')
```

Out[16]:

```
<generator object edits_two.<locals>.<genexpr> at 0x00000291B807ABF8>
```

In [17]:

```
def known(words):
    "The subset of `words` that appear in the `all_words`."
    return set(word for word in words if word in all_words)
```

In [18]:

```
def possible_corrections(word):
    "Generate possible spelling corrections for word."
    return (known([word]) or known(edits_one(word)) or known(edits_two(word)) or [word])
```

In [19]:

```
def prob(word, N=sum(all_words.values())):
    "Probability of `word`: Number of appearances of 'word' / total number of tokens"
    return all_words[word] / N
```

In [31]:

```
print(len(set(edits_one("monney"))))
print(edits_one("monney"))
```

```
336
{'money', 'monneyi', 'monsey', 'mowney', 'monkey', 'monner', 'monneyx', 'mox
nney', 'monnxy', 'menney', 'molney', 'lmonney', 'monneyb', 'smonney', 'monne
wy', 'monwney', 'morney', 'emonney', 'monneny', 'mznney', 'moniney', 'monnem
y', 'moenney', 'monnee', 'monnly', 'monuney', 'monnjey', 'monnoey', 'mtonne
y', 'mwonney', 'qonney', 'modney', 'mokney', 'monnzy', 'mzonney', 'monnej',
'dmonney', 'honney', 'monmey', 'ymonney', 'mobnney', 'mvonney', 'molnney',
'montey', 'monneay', 'mbnney', 'monnejy', 'mknney', 'mondey', 'monnley', 'mo
nnwey', 'monnyey', 'moonney', 'monnely', 'zmonney', 'mounney', 'monnvey', 'p
onney', 'mlnney', 'movney', 'mopnney', 'mosnney', 'ronney', 'qmonney', 'moqn
ey', 'monnoy', 'monaey', 'mdonney', 'moyney', 'mnnney', 'moneey', 'monndy',
'moanney', 'moneney', 'monneqy', 'monneyc', 'monnez', 'omnney', 'monnxey',
'monxney', 'monneoy', 'nonney', 'xonney', 'mronney', 'monneyh', 'monnny', 'd
onney', 'monhey', 'zonney', 'monnney', 'monnedy', 'monneyg', 'monnmey', 'mon
nen', 'monngey', 'monnky', 'mhonney', 'omonney', 'imonney', 'monneby', 'mkon
ney', 'mobney', 'monnfy', 'monnwy', 'monnem', 'monnet', 'mojnney', 'wonney',
'monjey', 'monneiy', 'monnaey', 'mownney', 'monneyw', 'uonney', 'mnney', 'mo
nnew', 'mozney', 'msonney', 'oonney', 'mhnney', 'monneh', 'rmonney', 'moinne
y', 'wmonney', 'monnbey', 'monnevy', 'monneyk', 'mogney', 'monbney', 'mdnne
y', 'moxney', 'monfey', 'monneg', 'momney', 'mnoney', 'monne', 'mynney', 'mo
nnel', 'movnney', 'monncy', 'monney', 'monnay', 'moqnney', 'monwey', 'monxe
y', 'mornney', 'motnney', 'monnry', 'monneyr', 'monneq', 'monnecy', 'monbe
y', 'ionney', 'monnep', 'mponney', 'monneys', 'monzney', 'monneyn', 'monnce
y', 'monnpey', 'lonney', 'monnqy', 'monnes', 'monneb', 'monnef', 'monneyo',
'gonney', 'mouney', 'monpey', 'moaney', 'vmonney', 'monnhey', 'monnec', 'hmo
nney', 'monvney', 'mjnney', 'msnney', 'moncney', 'mongey', 'monnsy', 'monnt
y', 'mojney', 'monkney', 'monnei', 'monnye', 'monneyu', 'minney', 'momnney',
'monoey', 'monneu', 'monnev', 'monrney', 'mgonney', 'monnesy', 'mononey', 'm
onneyf', 'aonney', 'monny', 'amonney', 'monneey', 'xmonney', 'monnrey', 'mon
nsey', 'muonney', 'mohnney', 'monneyd', 'monneym', 'moiney', 'monneyj', 'moc
ney', 'monnezy', 'moeney', 'munney', 'onney', 'monnby', 'mwnney', 'monntey',
'monnea', 'moncey', 'monhney', 'monniey', 'mpnney', 'monqney', 'monnuey', 'm
ohney', 'monqey', 'fmonney', 'monneyp', 'monnfey', 'monneyv', 'mvnney', 'pmo
nney', 'bonney', 'monnek', 'monley', 'mqonney', 'monnery', 'fonney', 'monie
y', 'monnefy', 'monnhy', 'monnjy', 'monnmy', 'monnqey', 'monnepy', 'mqnney',
'mcnney', 'mgnney', 'mrnney', 'monneky', 'mtnney', 'mondney', 'umonney', 'co
nney', 'meonney', 'monneyl', 'mmonney', 'mbonney', 'monneyz', 'mofney', 'mon
rey', 'monnzey', 'eonney', 'moknney', 'maonney', 'mocnney', 'mxonney', 'motn
ey', 'moznney', 'mnonney', 'monnvy', 'monnyy', 'konney', 'mfonney', 'monney
e', 'cmonney', 'monnkey', 'monvey', 'monnegy', 'monneuy', 'jmonney', 'vonne
y', 'mopney', 'monnpy', 'monniy', 'modnney', 'monned', 'monyney', 'monnex',
'monnuy', 'monnehy', 'monnety', 'monneya', 'mconney', 'monaney', 'jonney',
'manney', 'tmonney', 'monneo', 'tonney', 'monlney', 'mofnney', 'mooney', 'mo
nmney', 'kmonney', 'montney', 'mfnney', 'monndey', 'mognney', 'monneyy', 'mo
ngney', 'monneyq', 'mxnney', 'monfney', 'moynney', 'mosney', 'monpney', 'mon
jney', 'yonney', 'monyey', 'nmonney', 'sonney', 'monuey', 'bmonney', 'mionne
y', 'myonney', 'moneny', 'monngy', 'mjonney', 'mlonney', 'monzey', 'monney
t', 'gmonney', 'mmnney', 'monnexy', 'monsney'}
```

In [21]:

```
print(known(edits_one("monney")))
```

```
{'money', 'monkey'}
```

In [28]:

```
# Let's look at words that are two edits away
print(len(set(edits_two("emfasize"))))
print(known(edits_one("monney")))
```

```
90902
{'money', 'monkey'}
```

In [29]:

```
# Let's look at possible corrections of a word
print(possible_corrections("emfasize"))
```

```
{'emphasize'}
```

In [24]:

```
# Let's look at probability of a word
print(prob("money"))
print(prob("monkey"))
```

```
0.0002922233626303688
5.378344097491451e-06
```

In [25]:

```
def spell_check(word):
    "Print the most probable spelling correction for `word` out of all the `possible_correc
    correct_word = max(possible_corrections(word), key=prob)
    if correct_word != word:
        return "Did you mean " + correct_word + "?"
    else:
        return "Correct spelling."
```

In [26]:

```
# test spell check
print(spell_check("monney"))
```

```
Did you mean money?
```

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. It's just that there are more than one way to do things in Python. And it's always nice to know more than one way to perform the same task.

Now, the seed document 'big.txt' is nothing but a book. It's 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. Now, in the next part, Krishna demonstrates the use of edit distance in the spell corrector.

You just saw two functions. The edits_one() function and the edits_two() function. 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.

In the video present below, you will see how to filter out the valid English words from these lists by creating a separate function called 'known()' to do so.

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 functions 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:

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.

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

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.

In [33]:

```
from spell_corrector import rectify
correct = rectify("helllo")
print(correct)
```

```
hello
```

# 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 really manual process. To replace words in such manner, you would need to read the entire corpus and look for such terms.

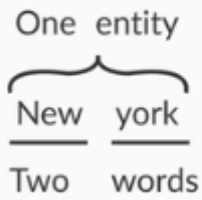
Turns out that there is a metric called the pointwise mutual information, also called the PMI. You can calculate the PMI score of each of these terms. PMI score of terms such as 'International Institute of Information Technology, Bangalore' will be much higher than other terms. If the PMI score is more than a certain threshold 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? In the following video, professor ME explains PMI

## ADVANCE TOKENISATION

1. Involves identifying separate terms and representing them as a single token
2. Terms such as 'New York', 'Indian Institute of Technology', 'Larsen & Toubro' must be represented by a single token instead of multiple ones

## ADVANCE TOKENISATION




$$\frac{P(\text{Mumbai, Delhi})}{P(\text{Mumbai})\,P(\text{Delhi})} < 1$$

## POINTWISE MUTUAL INFORMATION

$$pmi(x;y) = \log \frac{p(x,y)}{p(x)\,p(y)}$$

$$P(A_n, A_{n-1}, ..., A_1) = P(A_n \mid A_{n-1}, ..., A_1) \cdot P(A_{n-1} \mid A_{n-2}, ..., A_1) \cdot ..... P(A_1)$$

$$pmi(z,y,x) = \log\left(\frac{p(z \mid y,x)\,p(y \mid x)\,\cancel{p(x)}}{\cancel{p(x)}\;p(y)\;p(z)}\right)$$

$$P(A_n, A_{n-1}, ..., A_1) = P(A_n \mid A_{n-1}, ..., A_1) \cdot P(A_{n-1} \mid A_{n-2}, ..., A_1) \cdot ..... P(A_1)$$

$$pmi(z,y,x) = \log\left(\frac{p(z \mid y,x)\,p(y \mid x)}{p(y)\;p(z)}\right)$$

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))]

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

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

**OCCURRENCE CONTEXT**

1. The window to look at to calculate the probability of a word or phrase is called the occurrence context
2. Various choices for occurrence contexts:
   a. Entire document
   b. Paragraph
   c. Sentence

**OCCURRENCE CONTEXT**

$$p(\text{Mumbai}) = \frac{\text{Numer of sentencs containing 'Mumbai'}}{\text{Total number of sentencs}}$$

**OCCURRENCE CONTEXT**

S1 : Constraction of a new metro station state in New Delhi

S2 : New Delhi is the capital of India

S3 : The climate of Delhi is monsoon-influenced humid climate

$$p(\text{New Delhi}) = \frac{2}{3}$$

$$p(\text{New}) = \frac{2}{3}$$

$$p(\text{Delhi}) = \frac{3}{3}$$

$$\text{pmi (New Delhi)} = \frac{p(\text{New Delhi})}{p(\text{New})\, p(\text{Delhi})}$$

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.

-0.176

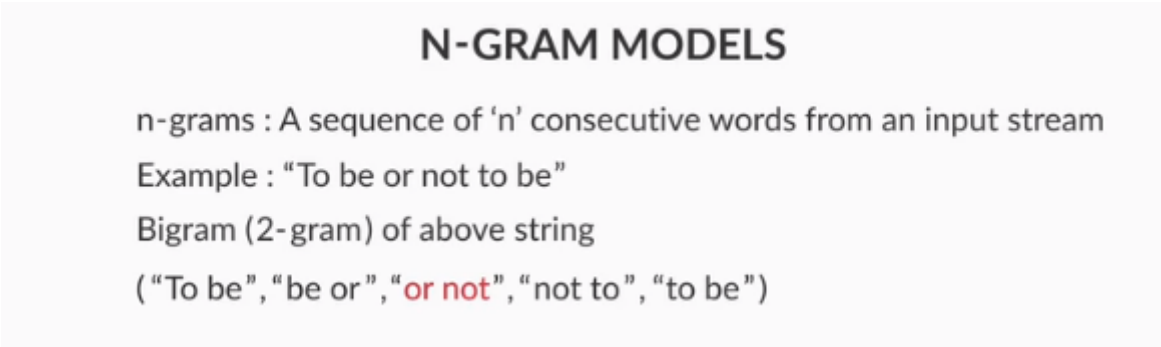
PMI(Nobel Prize) = log10(p(Nobel Prize)/(p(nobel)p(prize))).

Plugging in the values of p(Nobel Prize) as 0.5, p(nobel) as 1 and p(prize) as 0.75, you’ll get log10(0.67), i.e. -0.176 as the final PMI score.

## Pointwise Mutual Information - II

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.

Professor Me explains the chain rule in the following video.



## N-GRAM MODELS

Chain rule :

$P$("to be or not to be") =

$P$(to) $P$(be|to) $P$(or|"to be") $P$(not|"to be or") $P$(to|"to be or not") $P$(be|"to be or not to")

Bigram approximation :

$P$("to be or not to be") =

$P$(to) $P$(be|to) $P$(or|be) $P$(not|or) $P$(to|not) $P$(be|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 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.

This brings us to the end of the session. In the next section, you'll go through a quick summary of this session

# Summary

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

Handling differently spelt words due to different pronunciations

Correcting spelling of misspelt words using edit distance

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.

In [ ]: