**PART C**

**NAME-PRITAM BARUA ENROLLMENT-CST 510516042**

**UNDER THE GUIDANCE OF PROFESSOR JAYA SIL**

**A MINI PROJECT ON TEXT ANALYSIS**

|  |
| --- |
| **TOPICS COVERED** |
| **INTRODUCTION** |
| **STOP WORD REMOVAL** |
| **TRIE DATA STRUCTURE** |
| **SPELL CHECKER** |
| **FINDING OUT THE MOST RELEVANT STRING** |
| **OUTPUT** |
| **ALGORITHM** |
| **RESULTS** |
| **BIBLIOGRAPHY** |

**INTRODUCTION:**

The NLTK module is a massive tool kit, aimed at helping you with the entire Natural Language Processing (NLP) methodology. NLTK will aid you with everything from splitting sentences from paragraphs, splitting up words, recognizing the part of speech of those words, highlighting the main subjects, and then even with helping your machine to understand what the text is all about.

Natural language processing (NLP) is a field of computer science concerned with the interactions between computers and human languages, and, in particular, concerned with programming computers to fruitfully process large natural language data.

Text analysis involves systematic reading or observation of  texts which are assigned labels to indicate the presence of interesting, meaningful patterns. After labeling a large set of data, a researcher is able to statistically estimate the proportions of patterns in the texts, as well as correlations between patterns.

In this report I have presented the three basic things in my work:

From a given text-

1. **Preprocessing of the data by removal of stop words.**
2. **Finding related sentences by analyzing the user inputted query.**
3. **Create a spell checker.**
4. **Printing out the most appropriate sentence for a given query**.

**STOP WORD REMOVAL**

The idea of Natural Language Processing is to do some form of analysis, or processing, where the machine can understand, at least to some level, what the text means, says, or implies.This is an obviously massive challenge, but there are steps to doing it that anyone can follow. The main idea, however, is that computers simply do not, and will not, ever understand words directly. Humans don't either \*shocker\*. In humans, memory is broken down into electrical signals in the brain, in the form of neural groups that fire in patterns. There is a lot about the brain that remains unknown, but, the more we break down the human brain to the basic elements, we find out basic the elements really are. Well, it turns out computers store information in a very similar way! We need a way to get as close to that as possible if we're going to mimic how humans read and understand text. Generally, computers use numbers for everything, but we often see directly in programming where we use [binary signals[https://cdncache-a.akamaihd.net/items/it/img/arrow-10x10.png](https://www.pythonprogramming.net/stop-words-nltk-tutorial/?completed=/tokenizing-words-sentences-nltk-tutorial/#11877308)](https://www.pythonprogramming.net/stop-words-nltk-tutorial/?completed=/tokenizing-words-sentences-nltk-tutorial/#11877308) (True or False, which directly translate to 1 or 0, which originates directly from either the presence of an electrical signal (True, 1), or not (False, 0)). To do this, we need a way to convert words to values, in numbers, or signal patterns. The process of converting data to something a computer can understand is referred to as "pre-processing." One of the major forms of pre-processing is going to be filtering out useless data. In natural language processing, useless words (data), are referred to as stop words.

Immediately, we can recognize ourselves that some words carry more meaning than other words. We can also see that some words are just plain useless, and are filler words. We use them in the English language, for example, to sort of "fluff" up the sentence so it is not so strange sounding. An example of one of the most common, unofficial, useless words is the phrase "umm." People stuff in "umm" frequently, some more than others. This word means nothing, unless of course we're searching for someone who is maybe lacking confidence, is confused, or hasn't practiced much speaking. We all do it, you can hear me saying "umm" or "uhh" in the videos plenty of ...uh ... times. For most analysis, these words are useless

We would not want these words taking up space in our database, or taking up valuable processing time. As such, we call these words "stop words" because they are useless, and we wish to do nothing with them. Another version of the term "stop words" can be more literal: Words we stop on.

For example, you may wish to completely cease analysis if you detect words that are commonly used sarcastically, and stop immediately. Sarcastic words, or phrases are going to vary by lexicon and corpus. For now, we'll be considering stop words as words that just contain no meaning, and we want to remove them.

You can do this easily, by storing a list of words that you consider to be stop words. NLTK starts you off with a bunch of words that they consider to be stop words, you can access it via the NLTK corpus with

>>> set(stopwords.words('english'))  
{'ourselves', 'hers', 'between', 'yourself', 'but', 'again', 'there', 'about', 'once', 'during', 'out', 'very', 'having', 'with', 'they', 'own', 'an', 'be', 'some', 'for', 'do', 'its', 'yours', 'such', 'into', 'of', 'most', 'itself', 'other', 'off', 'is', 's', 'am', 'or', 'who', 'as', 'from', 'him', 'each', 'the', 'themselves', 'until', 'below', 'are', 'we', 'these', 'your', 'his', 'through', 'don', 'nor', 'me', 'were', 'her', 'more', 'himself', 'this', 'down', 'should', 'our', 'their', 'while', 'above', 'both', 'up', 'to', 'ours', 'had', 'she', 'all', 'no', 'when', 'at', 'any', 'before', 'them', 'same', 'and', 'been', 'have', 'in', 'will', 'on', 'does', 'yourselves', 'then', 'that', 'because', 'what', 'over', 'why', 'so', 'can', 'did', 'not', 'now', 'under', 'he', 'you', 'herself', 'has', 'just', 'where', 'too', 'only', 'myself', 'which', 'those', 'i', 'after', 'few', 'whom', 't', 'being', 'if', 'theirs', 'my', 'against', 'a', 'by', 'doing', 'it', 'how', 'further', 'was', 'here', 'than'}

Here is how you might incorporate using the stop\_words set to remove the stop words from your text:

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

example\_sent = "This is a sample sentence, showing off the stop words filtration."

stop\_words = set(stopwords.words('english'))

word\_tokens = word\_tokenize(example\_sent)

filtered\_sentence = [w for w in word\_tokens if not w in stop\_words]

filtered\_sentence = []

for w in word\_tokens:

if w not in stop\_words:

filtered\_sentence.append(w)

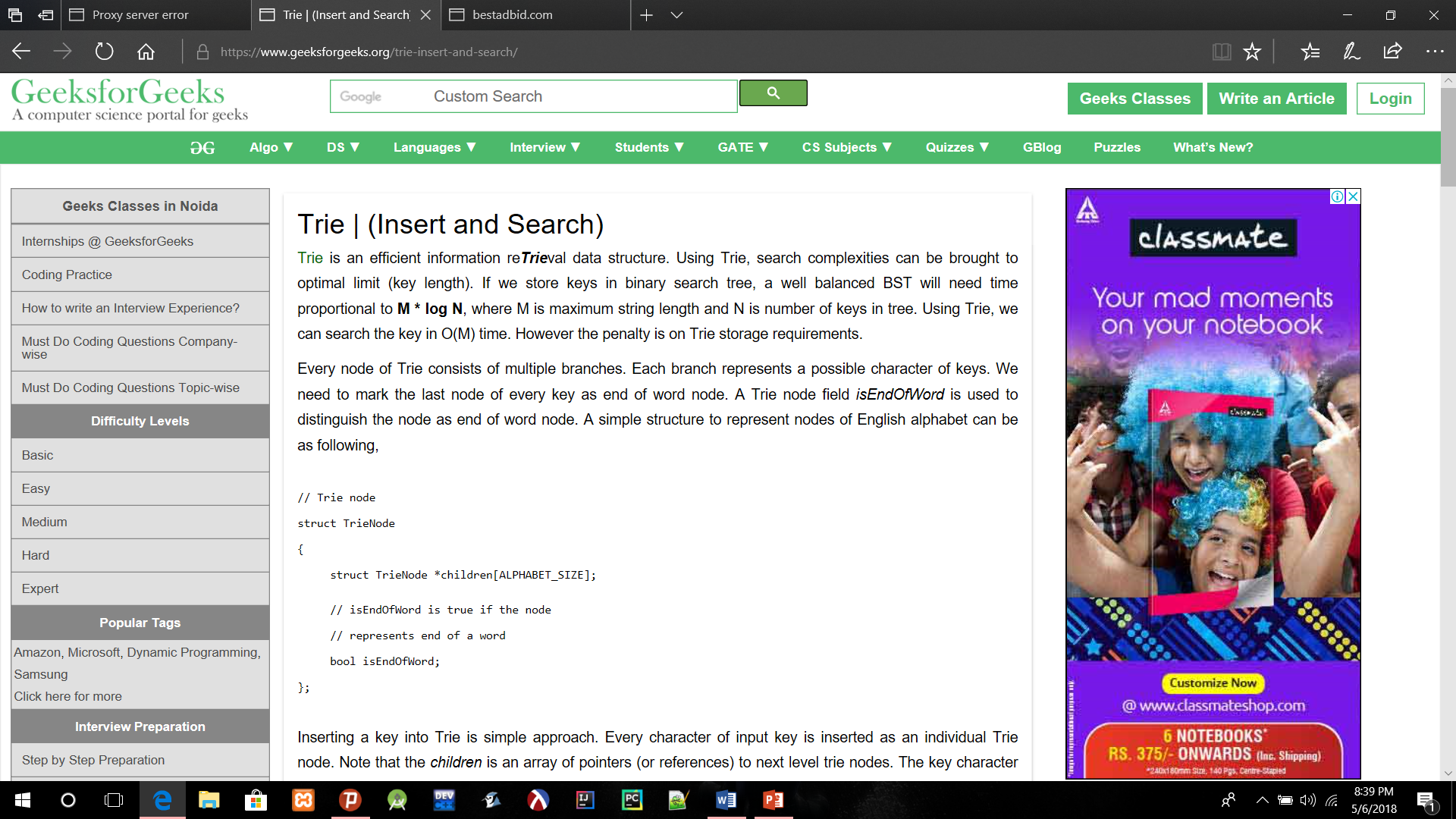
print(word\_tokens)

print(filtered\_sentence)

Our output here:  
['This', 'is', 'a', 'sample', 'sentence', ',', 'showing', 'off', 'the', 'stop', 'words', 'filtration', '.']  
['This', 'sample', 'sentence', ',', 'showing', 'stop', 'words', 'filtration', '.']

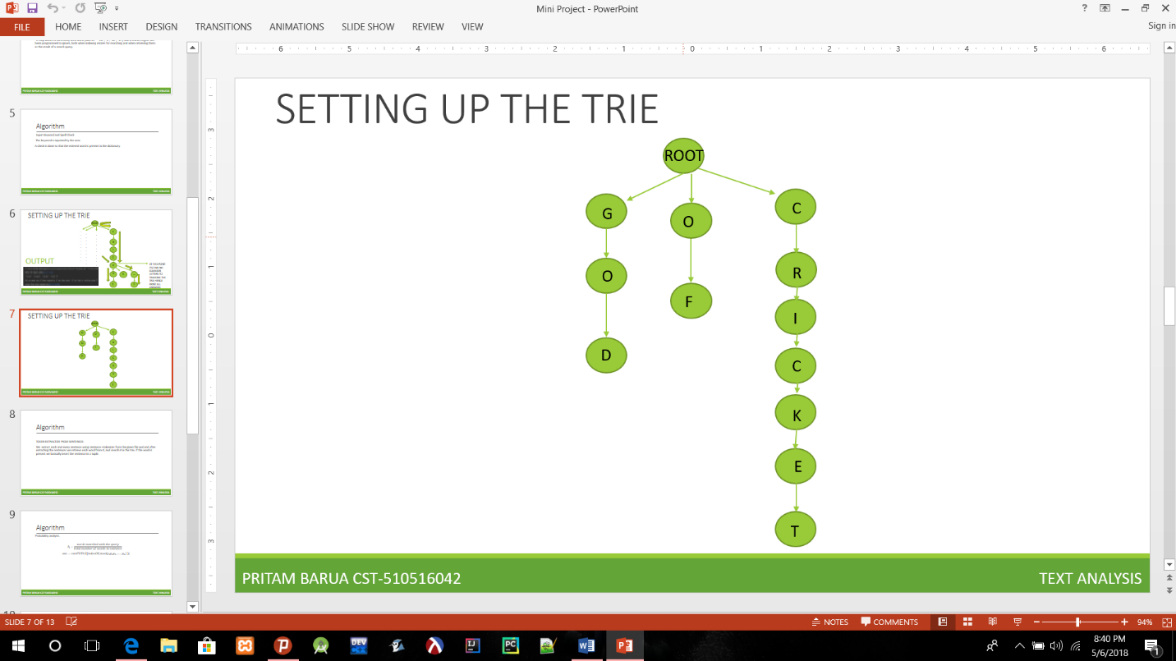
**TRIE DATA STRUCTURE**

***Trie***  data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to **M \* log N**, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in O(M) time. However the penalty is on Trie storage requirements.

Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node. A Trie node field *isEndOfWord* is used to distinguish the node as end of word node. A simple structure to represent nodes of English alphabet can be as following,  
  


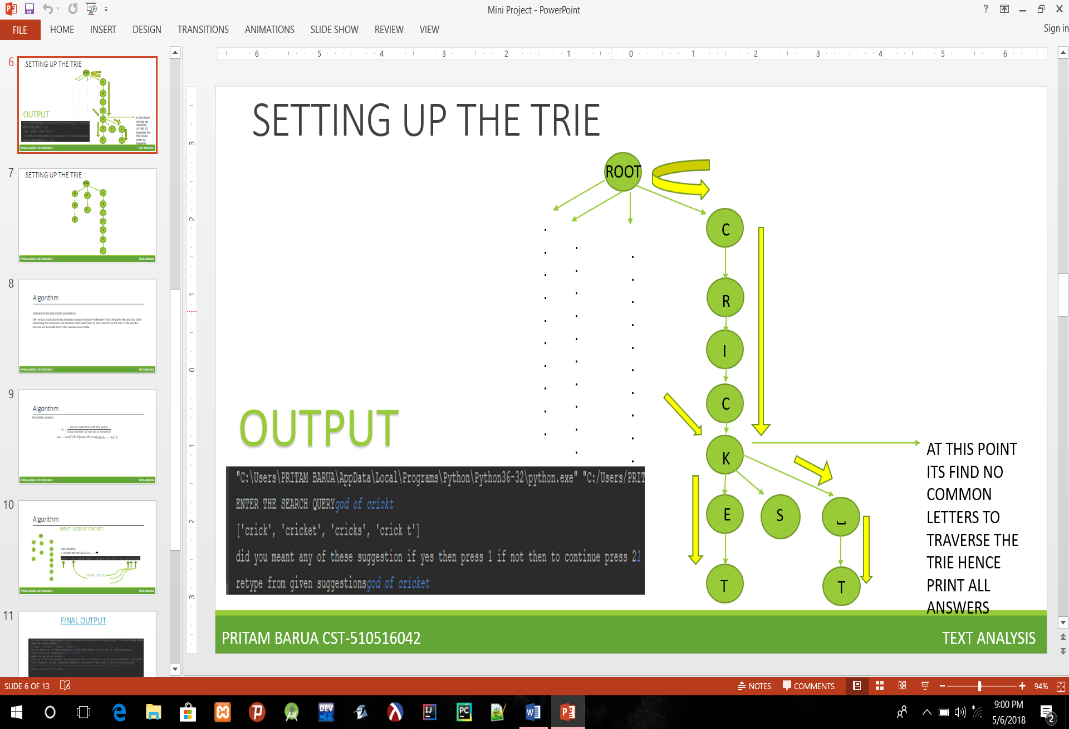
Inserting a key into Trie is simple approach. Every character of input key is inserted as an individual Trie node. Note that the *children* is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark end of word for last node. If the input key is prefix of existing key in Trie, we simply mark the last node of key as end of word. The key length determines Trie depth.

Searching for a key is similar to insert operation, however we only compare the characters and move down. The search can terminate due to end of string or lack of key in trie. In the former case, if the *isEndofWord* field of last node is true, then the key exists in trie. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.



**Spell checker**

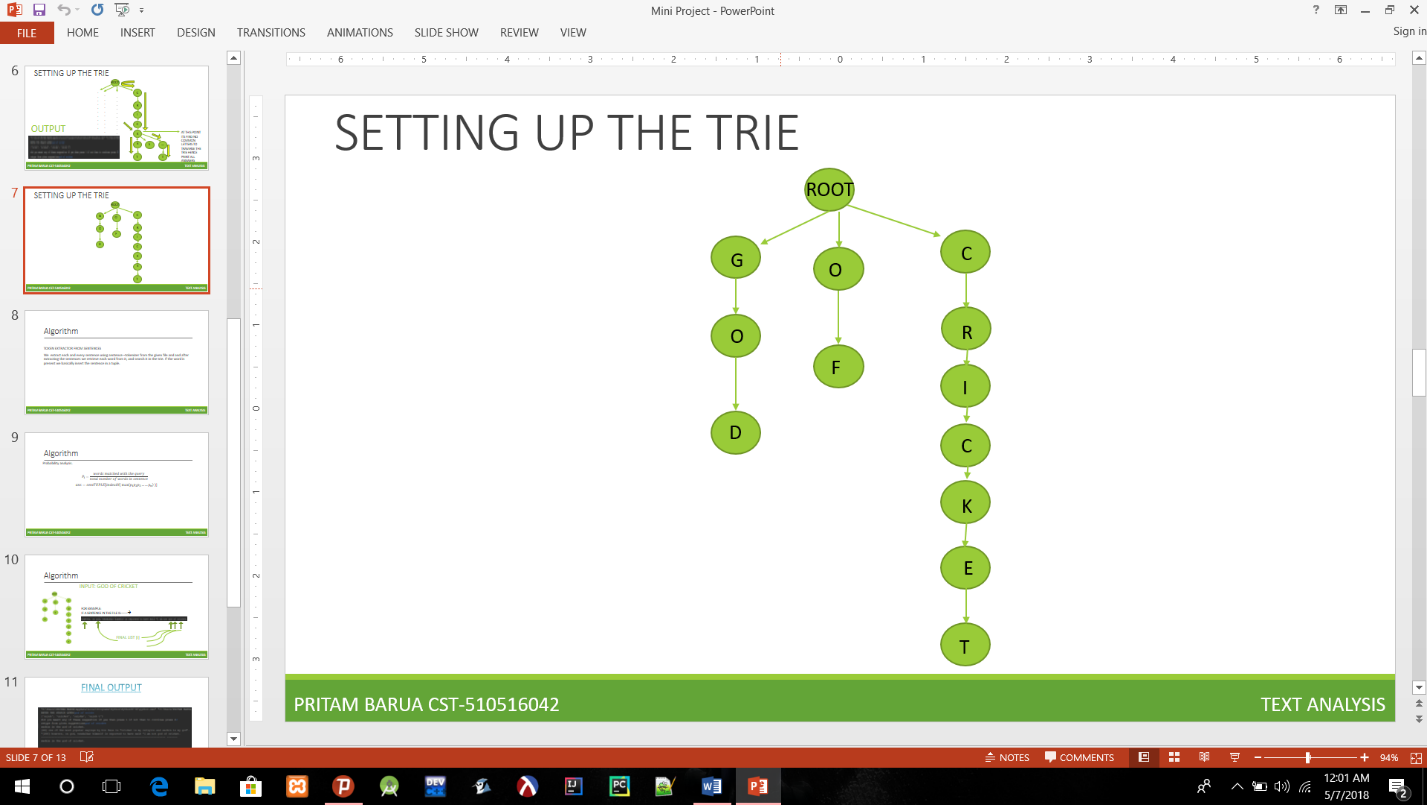
After we have finished setting up the entire trie data structure, we need to extract the given file and now tokenize the sentences . further we need to tokenize each word in the above sentence. Now we need to use the python dictionary and basically we feed the dictionary into the entire trie data structure, after we finish feeding it into the data structure we need to extract each word from the given query and then start to check whether that string is present in the given trie data structure. We have a copy pointer that keeps pointing and whenever we find that we have encoured something not present in the dictionary we give all the possibilities as a suggestion for the user that the given string can basically be these given alternatives.

The demo is :

We keep scanning each and every node one by one and change and shift our pointer according to that once we reach the point K we get T from the given query since no such word exist in the python dictionary then we go for CRICK-ET/S/ T. which are expected to be the probable outcomes.

**FINDING OUT THE MOST RELEVANT STRING**

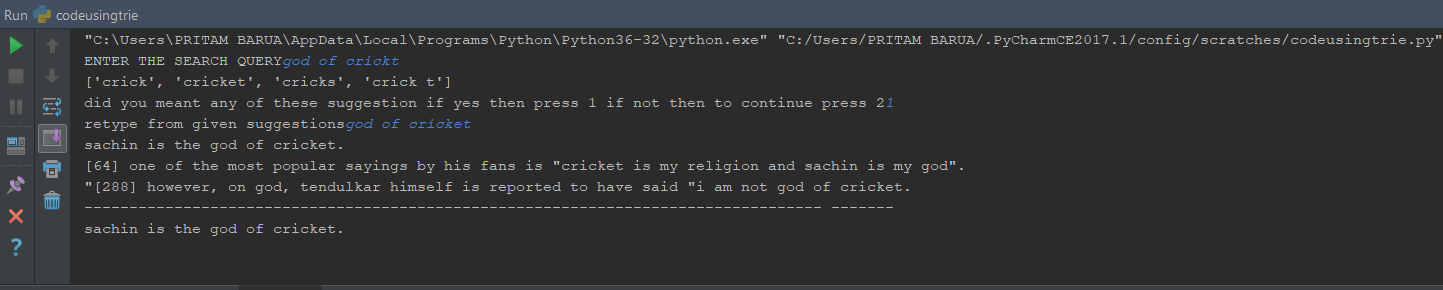
Out of all the sentence’s we have got our queries in, now we basically will find out the most relevant sentence out of all the sentences. At first, we need to set up the given query into the trie data structure as follow:

Once we finish doing this much of work, we basically store each of the sentences with matching query into a tuple. The process is simple we extract a token scan it inside a trie, if we find it to be true or present we add the sentence in the tuple sentTUPLE[].

In our given code we will basically count the number of distinct probable tokens from the strings present in the sentTUPLE. When we find it out now will calculate the probability of the most probable word, total no of occurrences of the given tokens divided by the total number of tokens in the sentences and store it in another tuple named and store the probabilities in the respective indexes. The sentence with the maximum probability number will be called as the most relevant string, I have used the formula:

***]***

And then finally we display the string stored in **ans.**

**OUTPUT OF THE CODE:**

**ALGORITHM**

**from** nltk.corpus **import** stopwords  
**from** nltk.tokenize **import** word\_tokenize  
**from** nltk.tokenize **import** sent\_tokenize  
**import** os  
**import** enchant  
**from** autocorrect **import** spell  
**from** datetime **import** datetime  
start=datetime.now()  
**class** TrieNode:  
 **def** \_\_init\_\_(self):  
 self.children = [**None**] \* 26  
 *# isEndOfWord is True if node represent the end of the word* self.isEndOfWord = **False  
class** Trie:  
 *# Trie data structure class* **def** \_\_init\_\_(self):  
 self.root = self.getNode()  
 **def** getNode(self):

*# Returns new trie node (initialized to NULLs)* **return** TrieNode()  
  
 **def** \_charToIndex(self, ch):  
 *# private helper function  
 # Converts key current character into index  
 # use only 'a' through 'z' and lower case* **return** ord(ch) - ord(**'a'**)  
  
 **def** insert(self, key):  
 *# If not present, inserts key into trie  
 # If the key is prefix of trie node,  
 # just marks leaf node* pCrawl = self.root  
 length = len(key)  
 **for** level **in** range(length):  
 index = self.\_charToIndex(key[level])  
 *# if current character is not present* **if not** pCrawl.children[index]:  
 pCrawl.children[index] = self.getNode()  
 pCrawl = pCrawl.children[index]  
 *# mark last node as leaf* pCrawl.isEndOfWord = **True  
 def** search(self, key):  
 *# Search key in the trie  
 # Returns true if key presents  
 # in trie, else false* pCrawl = self.root  
 length = len(key)  
 **for** level **in** range(length):  
 index = self.\_charToIndex(key[level])  
 **if not** pCrawl.children[index]:  
 **return False** pCrawl = pCrawl.children[index]  
 **return** pCrawl != **None and** pCrawl.isEndOfWord  
d = enchant.Dict()  
example\_sent = input(**"ENTER THE SEARCH QUERY"**)  
word\_tokens =[]  
stop\_words = set(stopwords.words(**'english'**))  
sent = []  
word\_token = word\_tokenize(example\_sent)  
**for** w **in** word\_token:  
 **if** (**not** d.check(w)):  
 print(d.suggest(w))  
 chk = input(**"did you meant any of these suggestion if yes then press 1 if not then to continue press 2"**)  
 **if** (chk **is '1'**):  
 example\_sent = input(**"retype from given suggestions"**)  
word\_tokens = word\_tokenize(example\_sent)  
filtered\_sentence = [w **for** w **in** word\_tokens **if not** w **in** stop\_words]  
tokensofquery = []  
Path = **"C:\\Users\PRITAM BARUA\.PyCharmCE2017.1\config\scratches\\"**filelist = os.listdir(Path)  
  
i=0  
**for** w **in** word\_tokens:  
 **if** w **not in** stop\_words:  
 tokensofquery.append(w)  
tokenascii = []  
**for** word **in** tokensofquery:  
 val = sum([ord(c) **for** c **in** word])  
 tokenascii.append(val)  
**for** i **in** filelist:  
 **if** i.endswith(**".txt"**):  
 file = open(i, **"r"**)  
 **with** file **as** fin:  
 tokensoffile = sent\_tokenize(fin.read())  
 c = 0  
 finallist = []  
 *# print(len(tokensofquery))* **for** w **in** tokensoffile:  
 word\_tokens = word\_tokenize(w)  
 hashed = [**"" for** x **in** range(1586)]  
 **for** k **in** word\_tokens:  
 val = sum([ord(c) **for** c **in** k])  
 *# print(k,val)* **if** (val <= 1586):  
 hashed[val] = k  
 c=0  
 **for** word **in** tokensofquery:  
 val = sum([ord(c) **for** c **in** word])  
 **if** (hashed[val] == word):  
 c = c + 1  
 **if** (c >= len(tokensofquery)):  
 finallist.append(w)  
  
 print(**"\n"**.join(finallist))  
  
  
 countfreq = []  
 **for** w **in** finallist:  
 word\_tokens = word\_tokenize(w)  
 count = 0  
 **for** word **in** tokensofquery:  
 favcases = 0  
  
 **for** k **in** word\_tokens:  
 **if** word == k:  
 favcases = favcases + 1  
 **if** favcases > 1:  
 favcases = 1  
 count = count + favcases  
 countfreq.append(count / len(w))  
 **if** len(countfreq):  
 max\_value = max(countfreq)  
 max\_index = countfreq.index(max\_value)  
 print(**"---------------------------------------------------------------------------------- -------"**)  
 print(finallist[max\_index])  
  
*#print(datetime.now()-start)*

RESULT

* We learnt the use of the basic python library functions as well as the NLTK library.
* We learnt the pre-processing of data like tokenizing, stop word removal, stemming, parts of speech tagging etc.
* We learnt about the possible way to find out the related words of a user entered keyword.
* Implementation of trie data structure.

**BIBLIOGRAPHY**

|  |
| --- |
| Python programming.net |
| Wikipedia.org |