**NLTK (Natural Language Toolkit)**

Natural Language Processing is manipulation or understanding text or speech by any software or machine and is that humans interact, understand each other views, and respond with the appropriate answer.

In NLP, this interaction, understanding, the response is made by a computer instead of a human.

NLTK stands for Natural Language Toolkit. This toolkit is one of the most powerful NLP libraries which contains packages to make machines understand human language and reply to it with an appropriate response. Tokenization, Stemming, Lemmatization, Punctuation, Character count, word count are some of these packages

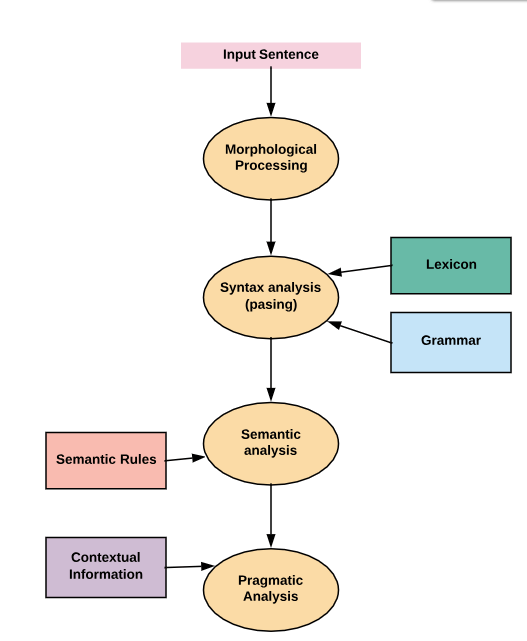
## Various NLP Libraries

|  |  |
| --- | --- |
| **NLP Library** | **Description** |
| NLTK | This is one of the most usable and mother of all NLP libraries. |
| spaCy | This is completely optimized and highly accurate library widely used in deep learning |
| Stanford CoreNLP Python | For client-server based architecture this is a good library in NLTK. This is written in JAVA, but it provides modularity to use it in Python. |
| TextBlob | This is an NLP library which works in Pyhton2 and python3. This is used for processing textual data and provide mainly all type of operation in the form of API. |
| Gensim | Genism is a robust open source NLP library support in python. This library is highly efficient and scalable. |
| Pattern | It is a light-weighted NLP module. This is generally used in Web-mining, crawling or such type of spidering task. P |
| Polyglot | For massive multilingual applications, Polyglot is best suitable NLP library. Feature extraction in the way on Identity and Entity. |
| PyNLPl | PyNLPI also was known as 'Pineapple' and supports Python. It provides a parser for many data format like FoLiA/Giza/Moses/ARPA/Timbl/CQL. |
| Vocabulary | This library is best to get Semantic type information from the given text. |

## **Components of NLP**

Five main Component of Natural Language processing are:

* Morphological and Lexical Analysis
* Syntactic Analysis
* Semantic Analysis
* Discourse Integration
* Pragmatic Analysis



**Morphological and Lexical Analysis**

Lexical analysis is a vocabulary that includes its words and expressions. It depicts analyzing, identifying and description of the structure of words. It includes dividing a text into paragraphs, words and the sentences

Individual words are analyzed into their components, and nonword tokens such as punctuations are separated from the words.

**Semantic Analysis**

Semantic Analysis is a structure created by the syntactic analyzer which assigns meanings. This component transfers linear sequences of words into structures. It shows how the words are associated with each other.

Semantics focuses only on the literal meaning of words, phrases, and sentences. This only abstracts the dictionary meaning or the real meaning from the given context. The structures assigned by the syntactic analyzer always have assigned meaning

E.g.. "colorless green idea." This would be rejected by the Symantec analysis as colorless Here; green doesn't make any sense.

**Pragmatic Analysis**

Pragmatic Analysis deals with the overall communicative and social content and its effect on interpretation. It means abstracting or deriving the meaningful use of language in situations. In this analysis, the main focus always on what was said in reinterpreted on what is meant.

Pragmatic analysis helps users to discover this intended effect by applying a set of rules that characterize cooperative dialogues.

E.g., "close the window?" should be interpreted as a request instead of an order.

**Syntax analysis**

The words are commonly accepted as being the smallest units of syntax. The syntax refers to the principles and rules that govern the sentence structure of any individual languages.

Syntax focus about the proper ordering of words which can affect its meaning. This involves analysis of the words in a sentence by following the grammatical structure of the sentence. The words are transformed into the structure to show hows the word are related to each other.

**Discourse Integration**

It means a sense of the context. The meaning of any single sentence which depends upon that sentences. It also considers the meaning of the following sentence.

For example, the word "that" in the sentence "He wanted that" depends upon the prior discourse context.

## NLP and writing systems

The kind of writing system used for a language is one of the deciding factors in determining the best approach for text pre-processing. Writing systems can be

1. Logographic: a Large number of individual symbols represent words. Example Japanese, Mandarin
2. Syllabic: Individual symbols represent syllables
3. Alphabetic: Individual symbols represent sound

Majority of the writing systems use the Syllabic or Alphabetic system. Even English, with its relatively simple writing system based on the Roman alphabet, utilizes logographic symbols which include Arabic numerals, Currency symbols (S, £), and other special symbols.

This pose following challenges

* Extracting meaning(semantics) from a text is a challenge
* NLP is dependent on the quality of the corpus. If the domain is vast, it's difficult to understand context.
* There is a dependence on the character set and language

## **How to implement NLP**

Below, given are popular methods used for Natural Learning Process:

**Machine learning & AI:**The learning nlp procedures used during machine learning. It automatically focuses on the most common cases. So when we write rules by hand, it is often not correct at all concerned about human errors.

**Statistical inference:**NLP can make use of statistical inference algorithms. It helps you to produce models that are robust. e.g., containing words or structures which are known to everyone.

## NLP Examples/Applications

Today, Natual process learning technology is widely used technology.

Here, are common Application' of NLP:

**Information retrieval & Web Search**

Google, Yahoo, Bing, and other search engines base their machine translation technology on NLP deep learning models. It allows algorithms to read text on a webpage, interpret its meaning and translate it to another language.

**Grammar Correction:**

NLP technique is widely used by word processor software like MS-word for spelling correction & grammar check.

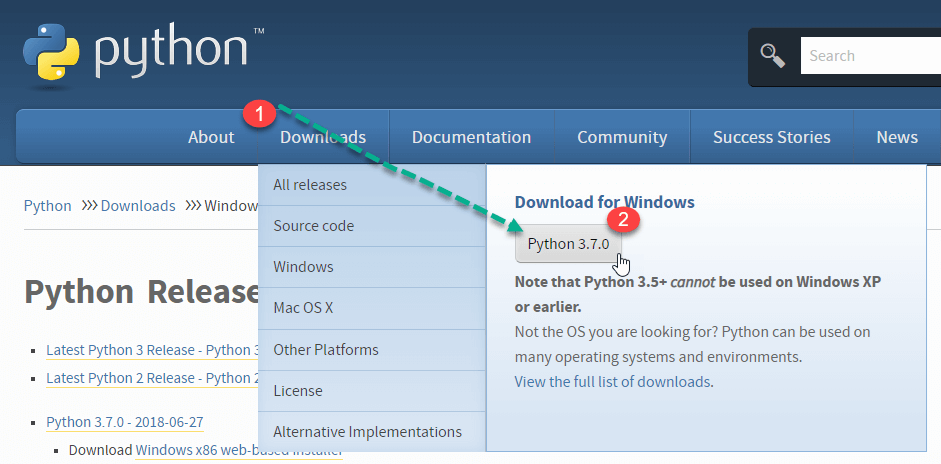
## Installing NLTK in Windows

In this part, we will learn that how to make setup NLTK via terminal (Command prompt in windows).

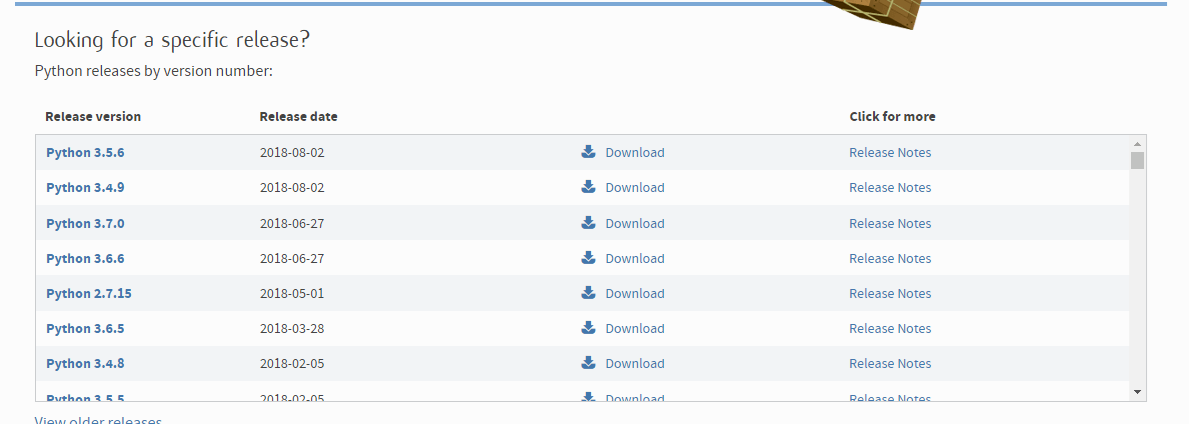
The instruction given below are based on the assumption that you don't have python installed. So, first step is to install python.

### Installing Python in Windows:

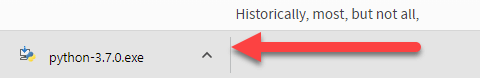
**Step 1)**Go to link **https://www.python.org/downloads/,**and select the latest version for windows.



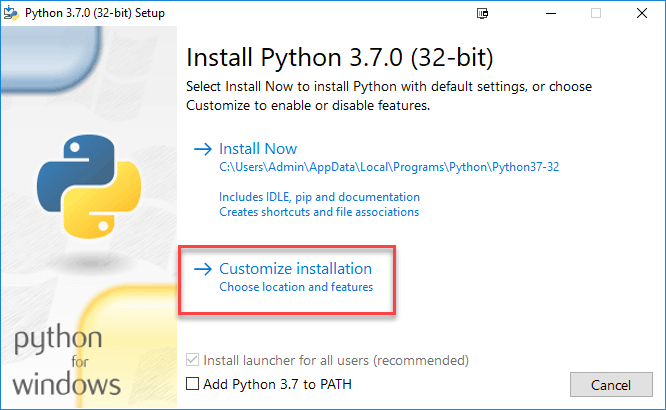
**Note**: If you don't want to download the latest version, you can visit the download tab and see all releases.



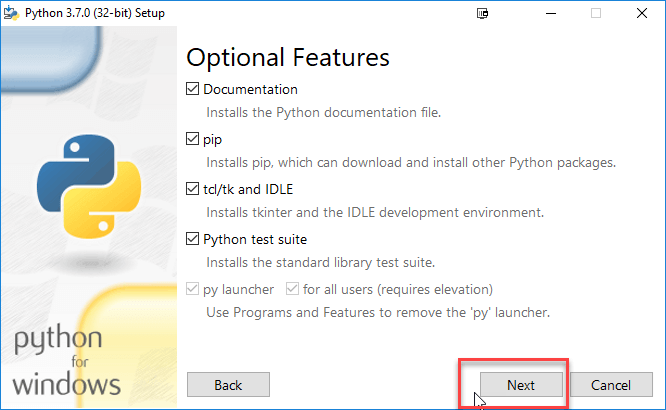
**Step 2)**Click on the Downloaded File



**Step 3)**Select Customize Installation

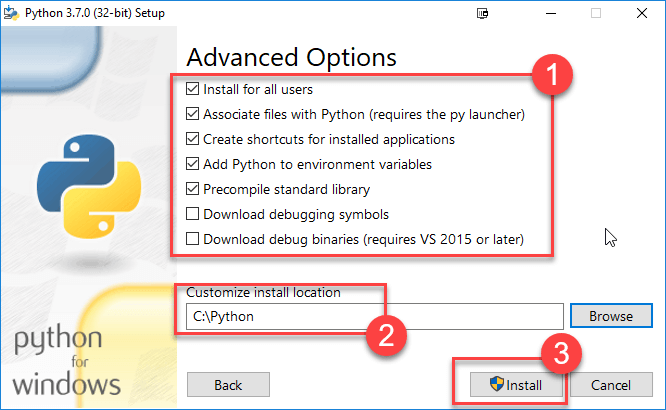


**Step 4)**Click NEXT

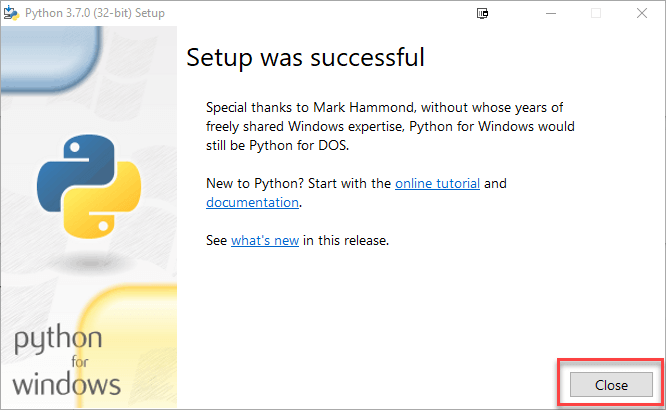


**Step 5)**In next screen

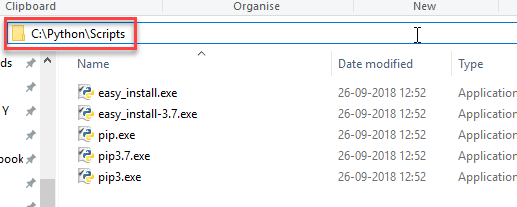
1. Select the advanced options
2. Give a Custom install location. In my case, a folder on C drive is chosen for ease in operation
3. Click Install



**Step 6)**Click Close button once install is done.



**Step 7)**Copy the path of your Scripts folder.

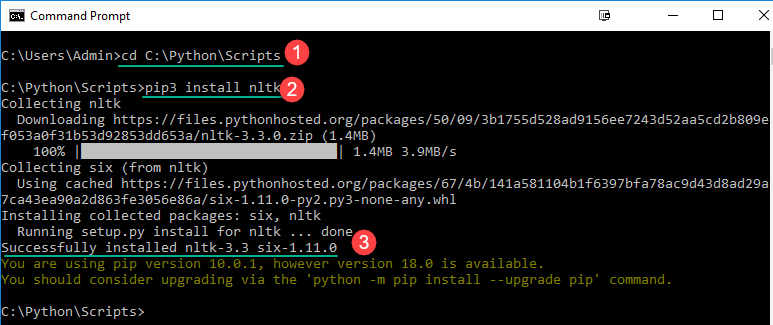


**Step 8)**In windows command prompt

* Navigate to the location of the pip folder
* Enter command to install NLTK

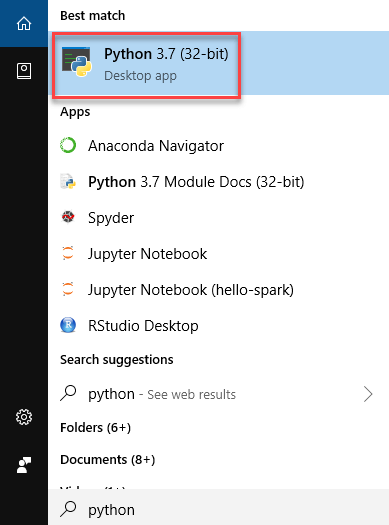
pip3 install nltk

* Installation should be done successfully



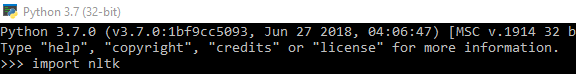
**NOTE**: For Python2 use the commandpip2 install nltk

**Step 9)**In Windows Start Menu, search and open PythonShell



**Step 10)**You can verify whether the installation is accurate supplying the below command

import nltk



If you see no error, Installation is complete.

## Installing NLTK in Mac/Linux

Installing NLTK in Mac/Unix requires python package manager pip to install nltk. If pip is not installed, please follow the below instructions to complete the process

**Step1)** Update the package index by typing the below command

sudo apt update

**Step2)** Installing pip for Python 3:

sudo apt install python3-pip

You can also install pip using easy\_install.

sudo apt-get install python-setuptools python-dev build-essential

Now easy\_install is installed. Run the below command to install pip

sudo easy\_install pip

**Step3)**Use following command to install NLTK

sudo pip install -U nltk

sudo pip3 install -U nltk

## How to Run NLTK Script

**Step1)**In your favorite code editor, copy the code and save the file as**"**NLTKsample.py**"**

from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer(r'\w+')

filterdText=tokenizer.tokenize('Hello Tech, You have build a very good site and I love visiting your site.')

print(filterdText)

## What is Tokenization?

Tokenization is the process by which big quantity of text is divided into smaller parts called **tokens**.

Natural language processing is used for building applications such as Text classification, intelligent chatbot, sentimental analysis, language translation, etc. It becomes vital to understand the pattern in the text to achieve the above-stated purpose. **These tokens are very useful for finding such patterns as well as is considered as a base step for stemming and lemmatization.**

For the time being, don't worry about stemming and lemmatization but treat them as steps for textual data cleaning using NLP (Natural language processing). We will discuss stemming and lemmatization later in the tutorial. Tasks such as **Text classification or spam filtering** makes use of NLP along with deep learning libraries such as Keras and Tensorflow.

Natural Language toolkit has very important module **tokenize** which further compromises of sub-modules

1. word tokenize
2. sentence tokenize

## Tokenization of words

We use the method **word\_tokenize()** to split a sentence into words. The output of word tokenization can be converted to Data Frame for better text understanding in machine learning applications. It can also be provided as input for further text cleaning steps such as punctuation removal, numeric character removal or stemming. Machine learning models need numeric data to be trained and make a prediction. Word tokenization becomes a crucial part of the text (string) to numeric data conversion. Please read about Bag of Words or CountVectorizer. Please refer to below example to understand the theory better.

from nltk.tokenize import word\_tokenize

text = "God is Great! I won a lottery."

print(word\_tokenize(text))

Output: ['God', 'is', 'Great', '!', 'I', 'won', 'a', 'lottery', '.']

1. word\_tokenize module is imported from the NLTK library.
2. A variable "text" is initialized with two sentences.
3. Text variable is passed in word\_tokenize module and printed the result. This module breaks each word with punctuation which you can see in the output.

## Tokenization of Sentences

Sub-module available for the above is sent\_tokenize. An obvious question in your mind would be **why sentence tokenization is needed when we have the option of word tokenization**. Imagine you need to count average words per sentence, how you will calculate? For accomplishing such a task, you need both sentence tokenization as well as words to calculate the ratio. Such output serves as an important feature for machine training as the answer would be numeric.

Check the below example to learn how sentence tokenization is different from words tokenization.

from nltk.tokenize import sent\_tokenize

text = "God is Great! I won a lottery."

print(sent\_tokenize(text))

Output: ['God is Great!', 'I won a lottery ']

We have **12**words and **two sentences** for the same input.

**Explanation of the program:**

1. In a line like the previous program, imported the sent\_tokenize module.
2. We have taken the same sentence. Further sent module parsed that sentences and show output. It is clear that this function breaks each sentence.

Above examples are good settings stones to understand the mechanics of the word and sentence tokenization.

# POS (Part-Of-Speech) Tagging & Chunking

## **POS Tagging**

Parts of speech Tagging is responsible for reading the text in a language and assigning some specific token (Parts of Speech) to each word.

e.g.

Input: Everything to permit us.

Output: [('Everything', NN),('to', TO), ('permit', VB), ('us', PRP)]

**Steps Involved:**

* Tokenize text (word\_tokenize)
* apply pos\_tag to above step that is nltk.pos\_tag(tokenize\_text)

**Some examples are as below:**

|  |  |
| --- | --- |
| **Abbreviation** | **Meaning** |
| CC | coordinating conjunction |
| CD | cardinal digit |
| DT | Determiner |
| EX | existential there |
| FW | foreign word |
| IN | preposition/subordinating conjunction |
| JJ | adjective (large) |
| JJR | adjective, comparative (larger) |
| JJS | adjective, superlative (largest) |
| LS | list market |
| MD | modal (could, will) |
| NN | noun, singular (cat, tree) |
| NNS | noun plural (desks) |
| NNP | proper noun, singular (sarah) |
| NNPS | proper noun, plural (indians or americans) |
| PDT | predeterminer (all, both, half) |
| POS | possessive ending (parent\ 's) |
| PRP | personal pronoun (hers, herself, him,himself) |
| PRP$ | possessive pronoun (her, his, mine, my, our ) |
| RB | adverb (occasionally, swiftly) |
| RBR | adverb, comparative (greater) |
| RBS | adverb, superlative (biggest) |
| RP | particle (about) |
| TO | infinite marker (to) |
| UH | interjection (goodbye) |
| VB | verb (ask) |
| VBG | verb gerund (judging) |
| VBD | verb past tense (pleaded) |
| VBN | verb past participle (reunified) |
| VBP | verb, present tense not 3rd person singular(wrap) |
| VBZ | verb, present tense with 3rd person singular (bases) |
| WDT | wh-determiner (that, what) |
| WP | wh- pronoun (who) |
| WRB | wh- adverb (how) |

POS tagger is used to assign grammatical information of each word of the sentence. Installing, Importing and downloading all the packages of NLTK is complete.

## Chunking

Chunking is used to add more structure to the sentence by following parts of speech (POS) tagging. It is also known as shallow parsing. The resulted group of words is called "**chunks**." In shallow parsing, there is maximum one level between roots and leaves while deep parsing comprises of more than one level. Shallow Parsing is also called light parsing or chunking.

The primary usage of chunking is to make a group of "noun phrases." The parts of speech are combined with regular expressions.

**Rules for Chunking:**

There are no pre-defined rules, but you can combine them according to need and requirement.

For example, you need to tag Noun, verb (past tense), adjective, and coordinating junction from the sentence. You can use the rule as below

chunk:{<NN.?>\*<VBD.?>\*<JJ.?>\*<CC>?}

Following table shows what the various symbol means:

|  |  |
| --- | --- |
| **Name of symbol** | **Description** |
| . | Any character except new line |
| \* | Match 0 or more repetitions |
| ? | Match 0 or 1 repetitions |

Now Let us write the code to understand rule better

from nltk import pos\_tag

from nltk import RegexpParser

text ="learn php from Tech and make study easy".split()

print("After Split:",text)

tokens\_tag = pos\_tag(text)

print("After Token:",tokens\_tag)

patterns= """mychunk:{<NN.?>\*<VBD.?>\*<JJ.?>\*<CC>?}"""

chunker = RegexpParser(patterns)

print("After Regex:",chunker)

output = chunker.parse(tokens\_tag)

print("After Chunking",output)

**Output**

After Split: ['learn', 'php', 'from', 'Tech', 'and', 'make', 'study', 'easy']

After Token: [('learn', 'JJ'), ('php', 'NN'), ('from', 'IN'), ('Tech', 'NN'), ('and', 'CC'), ('make', 'VB'), ('study', 'NN'), ('easy', 'JJ')]

After Regex: chunk.RegexpParser with 1 stages:

RegexpChunkParser with 1 rules:

<ChunkRule: '<NN.?>\*<VBD.?>\*<JJ.?>\*<CC>?'>

After Chunking (S

(mychunk learn/JJ)

(mychunk php/NN)

from/IN

(mychunk Tech/NN and/CC)

make/VB

(mychunk study/NN easy/JJ))

The conclusion from the above example: "make" is a verb which is not included in the rule, so it is not tagged as mychunk

#### Use Case of Chunking

Chunking is used for entity detection. An entity is that part of the sentence by which machine get the value for any intention

Example:

Temperature of New York.

Here Temperature is the intention and New York is an entity.

In other words, chunking is used as selecting the subsets of tokens. Please follow the below code to understand how chunking is used to select the tokens. In this example, you will see the graph which will correspond to a chunk of a noun phrase. We will write the code and draw the graph for better understanding.

### Code to Demonstrate Use Case

import nltk

text = "learn php from Tech"

tokens = nltk.word\_tokenize(text)

print(tokens)

tag = nltk.pos\_tag(tokens)

print(tag)

grammar = "NP: {<DT>?<JJ>\*<NN>}"

cp =nltk.RegexpParser(grammar)

result = cp.parse(tag)

print(result)

result.draw() # It will draw the pattern graphically which can be seen in Noun Phrase chunking

Output:

['learn', 'php', 'from', 'Tech'] -- These are the tokens

[('learn', 'JJ'), ('php', 'NN'), ('from', 'IN'), ('Tech', 'NN')] -- These are the pos\_tag

(S (NP learn/JJ php/NN) from/IN (NP Tech/NN)) -- Noun Phrase Chunking

**Noun Phrase chunking Graph**

From the graph, we can conclude that "learn" and "Tech" are two different tokens but are categorized as Noun Phrase whereas token "from" does not belong to Noun Phrase.

# Stemming and Lemmatization

## What is Stemming?

Stemming is a kind of normalization for words. Normalization is a technique where a set of words in a sentence are converted into a sequence to shorten its lookup. The words which have the same meaning but have some variation according to the context or sentence are normalized.

In another word, there is one root word, but there are many variations of the same words. For example, the root word is "eat" and it's variations are "eats, eating, eaten and like so". In the same way, with the help of Stemming, we can find the root word of any variations.

**For example**

**Program for understanding Stemming**

from nltk.stem import PorterStemmer

e\_words= ["wait", "waiting", "waited", "waits"]

ps =PorterStemmer()

for w in e\_words:

rootWord=ps.stem(w)

print(rootWord)

**Output**:

wait

wait

wait

wait

**Code Explanation:**

* There is a stem module in NLTk which is imported. If ifyou import the complete module, then the program becomes heavy as it contains thousands of lines of codes. So from the entire stem module, we only imported "PorterStemmer."
* We prepared a dummy list of variation data of the same word.
* An object is created which belongs to class nltk.stem.porter.PorterStemmer.
* Further, we passed it to PorterStemmer one by one using "for" loop. Finally, we got output root word of each word mentioned in the list.

From the above explanation, it can also be concluded that stemming is considered as an important preprocessing step because it removed redundancy in the data and variations in the same word. As a result, data is filtered which will help in better machine training.

Now we pass a complete sentence and check for its behavior as an output.

**Program:**

from nltk.stem import PorterStemmer

from nltk.tokenize import sent\_tokenize, word\_tokenize

sentence="Hello Tech, You have to build a very good site and I love visiting your site."

words = word\_tokenize(sentence)

ps = PorterStemmer()

for w in words:

rootWord=ps.stem(w)

print(rootWord)

**Output:**

hello

Tech

,

you

have

build

a

veri

good

site

and

I

love

visit

your

site

## **What is Lemmatization?**

Lemmatization is the algorithmic process of finding the lemma of a word depending on their meaning. Lemmatization usually refers to the morphological analysis of words, which aims to remove inflectional endings. It helps in returning the base or dictionary form of a word, which is known as the lemma. The NLTK Lemmatization method is based on WorldNet's built-in morph function. Text preprocessing includes both stemming as well as lemmatization. Many people find the two terms confusing. Some treat these as same, but there is a difference between these both. Lemmatization is preferred over the former because of the below reason.

## **Why is Lemmatization better than Stemming?**

Stemming algorithm works by cutting the suffix from the word. In a broader sense cuts either the beginning or end of the word.

On the contrary, Lemmatization is a more powerful operation, and it takes into consideration morphological analysis of the words. It returns the lemma which is the base form of all its inflectional forms. In-depth linguistic knowledge is required to create dictionaries and look for the proper form of the word. Stemming is a general operation while lemmatization is an intelligent operation where the proper form will be looked in the dictionary. Hence, lemmatization helps in forming better machine learning features.

## Code to distinguish between Lemmatization and Stemming

Stemming code

import nltk

from nltk.stem.porter import PorterStemmer

porter\_stemmer = PorterStemmer()

text = "studies studying cries cry"

tokenization = nltk.word\_tokenize(text)

for w in tokenization:

print("Stemming for {} is {}".format(w,porter\_stemmer.stem(w)))

Output:

Stemming for studies is studi

Stemming for studying is studi

Stemming for cries is cri

Stemming for cry is cri

**Lemmatization code**

import nltk

from nltk.stem import WordNetLemmatizer

wordnet\_lemmatizer = WordNetLemmatizer()

text = "studies studying cries cry"

tokenization = nltk.word\_tokenize(text)

for w in tokenization:

print("Lemma for {} is {}".format(w, wordnet\_lemmatizer.lemmatize(w)))

Output:

Lemma for studies is study

Lemma for studying is studying

Lemma for cries is cry

Lemma for cry is cry

## **Use Case of Lemmatizer:**

Lemmatizer minimizes text ambiguity. Example words like bicycle or bicycles are converted to base word bicycle. Basically, it will convert all words having the same meaning but different representation to their base form. It reduces the word density in the given text and helps in preparing the accurate features for training machine. Cleaner the data, the more intelligent and accurate your machine learning model, will be. Lemmatizerwill also saves memory as well as computational cost.

**Real Time example showing use of Wordnet Lemmatization and POS Tagging in Python**

from nltk.corpus import wordnet as wn

from nltk.stem.wordnet import WordNetLemmatizer

from nltk import word\_tokenize, pos\_tag

from collections import defaultdict

tag\_map = defaultdict(lambda : wn.NOUN)

tag\_map['J'] = wn.ADJ

tag\_map['V'] = wn.VERB

tag\_map['R'] = wn.ADV

text = "Tech is a totally new kind of learning experience."

tokens = word\_tokenize(text)

lemma\_function = WordNetLemmatizer()

for token, tag in pos\_tag(tokens):

lemma = lemma\_function.lemmatize(token, tag\_map[tag[0]])

print(token, "=>", lemma)

**Code Explanation**

* Firstly, the corpus reader wordnet is imported.
* WordNetLemmatizer is imported from wordnet
* Word tokenize as well as parts of speech tag are imported from nltk
* Default Dictionary is imported from collections
* Dictionary is created where pos\_tag (first letter) are the key values whose values are mapped with the value from wordnet dictionary. We have taken the only first letter as we will use it later in the loop.
* Text is written and is tokenized.
* Object lemma\_function is created which will be used inside the loop
* Loop is run and lemmatize will take two arguments one is token and other is a mapping of pos\_tag with wordnet value.

Output:

Tech => Tech

is => be

totally => totally

new => new

kind => kind

of => of

learning => learn

experience => experience

. => .

# WordNet with NLTK: Finding Synonyms for words

## What is Wordnet?

Wordnet is an NLTK corpus reader, a lexical database for English. It can be used to find the meaning of words, synonym or antonym. One can define it as a semantically oriented dictionary of English. It is imported with the following command:

from nltk.corpus import wordnet as tech

Stats reveal that there are **155287 words and 117659 synonym** sets included with English WordNet.

Different methods available with WordNet can be found by typing dir(tech)

['\_LazyCorpusLoader\_\_args', '\_LazyCorpusLoader\_\_kwargs', '\_LazyCorpusLoader\_\_load', '\_LazyCorpusLoader\_\_name', '\_LazyCorpusLoader\_\_reader\_cls', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattr\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_name\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_unicode\_\_', '\_\_weakref\_\_', '\_unload', 'subdir', 'unicode\_repr']

Let us understand some of the features available with the wordnet:

**Synset**: It is also called as synonym set or collection of synonym words. Let us check a example

from nltk.corpus import wordnet

syns = wordnet.synsets("dog")

print(syns)

Output:

[Synset('dog.n.01'), Synset('frump.n.01'), Synset('dog.n.03'), Synset('cad.n.01'), Synset('frank.n.02'), Synset('pawl.n.01'), Synset('andiron.n.01'), Synset('chase.v.01')]

**Lexical Relations**: These are semantic relations which are reciprocated. If there is a relationship between {x1,x2,...xn} and {y1,y2,...yn} then there is also relation between {y1,y2,...yn} and {x1,x2,...xn}. For example Synonym is the opposite of antonym or hypernyms and hyponym are type of lexical concept.

Let us write a program using python to find synonym and antonym of word "active" using Wordnet.

from nltk.corpus import wordnet

synonyms = []

antonyms = []

for syn in wordnet.synsets("active"):

for l in syn.lemmas():

synonyms.append(l.name())

if l.antonyms():

antonyms.append(l.antonyms()[0].name())

print(set(synonyms))

print(set(antonyms))

{'dynamic', 'fighting', 'combat-ready', 'active\_voice', 'active\_agent', 'participating', 'alive', 'active'} -- Synonym

{'stative', 'passive', 'quiet', 'passive\_voice', 'extinct', 'dormant', 'inactive'} -- Antonym

# Counting POS Tags, Frequency Distribution & Collocations

## COUNTING POS TAGS

We have discussed various **pos\_tag** in the previous section. In this particular tutorial, you will study how to count these tags. Counting tags are crucial for text classification as well as preparing the features for the Natural language-based operations. I will be discussing with you the approach which Tech followed while preparing code along with a discussion of output. Hope this will help you.

How to count Tags:

Here first we will write working code and then we will write different steps to explain the code.

from collections import Counter

import nltk

text = " Tech is one of the best sites to learn WEB, SAP, Ethical Hacking and much more online."

lower\_case = text.lower()

tokens = nltk.word\_tokenize(lower\_case)

tags = nltk.pos\_tag(tokens)

counts = Counter( tag for word, tag in tags)

print(counts)

**Output:**

Counter({'NN': 5, ',': 2, 'TO': 1, 'CC': 1, 'VBZ': 1, 'NNS': 1, 'CD': 1, '.': 1, 'DT': 1, 'JJS': 1, 'JJ': 1, 'JJR': 1, 'IN': 1, 'VB': 1, 'RB': 1})

1. To count the tags, you can use the package Counter from the collection's module. A counter is a dictionary subclass which works on the principle of key-value operation. It is an unordered collection where elements are stored as a dictionary key while the count is their value.
2. Import nltk which contains modules to tokenize the text.
3. Write the text whose pos\_tag you want to count.
4. Some words are in upper case and some in lower case, so it is appropriate to transform all the words in the lower case before applying tokenization.
5. Pass the words through word\_tokenize from nltk.
6. Calculate the pos\_tag of each token

Output = [('Tech', 'NN'), ('is', 'VBZ'), ('one', 'CD'), ('of', 'IN'), ('the', 'DT'), ('best', 'JJS'), ('site', 'NN'), ('to', 'TO'), ('learn', 'VB'), ('web', 'NN'), (',', ','), ('sap', 'NN'), (',', ','), ('ethical', 'JJ'), ('hacking', 'NN'), ('and', 'CC'), ('much', 'RB'), ('more', 'JJR'), ('online', 'JJ')]

1. Now comes the role of dictionary counter. We have imported in the code line 1. Words are the key and tags are the value and counter will count each tag total count present in the text.

## Frequency Distribution

Frequency Distribution is referred to as the number of times an outcome of an experiment occurs. It is used to find the frequency of each word occurring in a document. It uses **FreqDistclass** and defined by **the nltk.probabilty**module.

A frequency distribution is usually created by counting the samples of repeatedly running the experiment. The no of counts is incremented by one, each time. E.g.

freq\_dist = FreqDist()

for the token in the document:

freq\_dist.inc(token.type())

For any word, we can check how many times it occurred in a particular document. E.g.

1. **Count Method:**freq\_dist.count('and')This expression returns the value of the number of times 'and' occurred. It is called the count method.
2. **Frequency Method:**freq\_dist.freq('and')This the expression returns frequency of a given sample.

We will write a small program and will explain its working in detail. We will write some text and will calculate the frequency distribution of each word in the text.

import nltk

a = "Tech is the site where you can find the best tutorials for Software Testing Tutorial, SAP Course for Beginners. Java Tutorial for Beginners and much more. Please visit the site Tech.com and much more."

words = nltk.tokenize.word\_tokenize(a)

fd = nltk.FreqDist(words)

fd.plot()

**Explanation of code:**

1. Import nltk module.
2. Write the text whose word distribution you need to find.
3. Tokenize each word in the text which is served as input to FreqDist module of the nltk.
4. Apply each word to nlk.FreqDist in the form of a list
5. Plot the words in the graph using plot()

### Bigrams Example Code

import nltk

text = "Tech is a totally new kind of learning experience."

Tokens = nltk.word\_tokenize(text)

output = list(nltk.bigrams(Tokens))

print(output)

Output

[('Tech', 'is'), ('is', 'totally'), ('totally', 'new'), ('new', 'kind'), ('kind', 'of'), ('of', 'learning'), ('learning', 'experience'), ('experience', '.')]

### Trigrams Example Code

Sometimes it becomes important to see a pair of three words in the sentence for statistical analysis and frequency count. This again plays a crucial role in forming NLP (natural language processing features) as well as text-based sentimental prediction.

The same code is run for calculating the trigrams.

import nltk

text = “Tech is a totally new kind of learning experience.”

Tokens = nltk.word\_tokenize(text)

output = list(nltk.trigrams(Tokens))

print(output)

Output

[('Tech', 'is', 'totally'), ('is', 'totally', 'new'), ('totally', 'new', 'kind'), ('new', 'kind', 'of'), ('kind', 'of', 'learning'), ('of', 'learning', 'experience'), ('learning', 'experience', '.')]

**Project**

## **Import Statments**

In [1]:

import numpy as np

import pandas as pd

from sklearn import preprocessing

from sklearn.feature\_extraction.text import CountVectorizer, TfidfVectorizer

from sklearn.model\_selection import train\_test\_split, KFold

from nltk.corpus import stopwords

from nltk.stem.snowball import SnowballStemmer

import matplotlib

from matplotlib import pyplot as plt

*# import seaborn as sns*

%matplotlib inline

%config InlineBackend.figure\_format = 'retina'

# ****Loading and inspecting data****

Reading the data

In [2]:

data = pd.read\_csv("../input/train.csv")

Displaying the head of the data

In [3]:

data.head(10)

Out[3]:

|  | id | text | author |
| --- | --- | --- | --- |
| 0 | id26305 | This process, however, afforded me no means of... | EAP |
| 1 | id17569 | It never once occurred to me that the fumbling... | HPL |
| 2 | id11008 | In his left hand was a gold snuff box, from wh... | EAP |
| 3 | id27763 | How lovely is spring As we looked from Windsor... | MWS |
| 4 | id12958 | Finding nothing else, not even gold, the Super... | HPL |
| 5 | id22965 | A youth passed in solitude, my best years spen... | MWS |
| 6 | id09674 | The astronomer, perhaps, at this point, took r... | EAP |
| 7 | id13515 | The surcingle hung in ribands from my body. | EAP |
| 8 | id19322 | I knew that you could not say to yourself 'ste... | EAP |
| 9 | id00912 | I confess that neither the structure of langua... | MWS |

Shape of the data

In [4]:

data.shape

Out[4]:

(19579, 3)

**Bar chart of class proportion**

In [5]:

*# extracting the number of examples of each class*

EAP\_len = data[data['author'] == 'EAP'].shape[0]

HPL\_len = data[data['author'] == 'HPL'].shape[0]

MWS\_len = data[data['author'] == 'MWS'].shape[0]

In [6]:

*# bar plot of the 3 classes*

plt.bar(10,EAP\_len,3, label="EAP")

plt.bar(15,HPL\_len,3, label="HPL")

plt.bar(20,MWS\_len,3, label="MWS")

plt.legend()

plt.ylabel('Number of examples')

plt.title('Propoertion of examples')

plt.show()

# ****Feature Engineering****

## **Removing punctions**

**Funtion to remove punctuation**

In [7]:

def remove\_punctuation(text):

*'''a function for removing punctuation'''*

import string

*# replacing the punctuations with no space,*

*# which in effect deletes the punctuation marks*

translator = str.maketrans('', '', string.punctuation)

*# return the text stripped of punctuation marks*

return text.translate(translator)

**Apply the function to each examples**

In [8]:

data['text'] = data['text'].apply(remove\_punctuation)

data.head(10)

Out[8]:

|  | id | text | author |
| --- | --- | --- | --- |
| 0 | id26305 | This process however afforded me no means of a... | EAP |
| 1 | id17569 | It never once occurred to me that the fumbling... | HPL |
| 2 | id11008 | In his left hand was a gold snuff box from whi... | EAP |
| 3 | id27763 | How lovely is spring As we looked from Windsor... | MWS |
| 4 | id12958 | Finding nothing else not even gold the Superin... | HPL |
| 5 | id22965 | A youth passed in solitude my best years spent... | MWS |
| 6 | id09674 | The astronomer perhaps at this point took refu... | EAP |
| 7 | id13515 | The surcingle hung in ribands from my body | EAP |
| 8 | id19322 | I knew that you could not say to yourself ster... | EAP |
| 9 | id00912 | I confess that neither the structure of langua... | MWS |

## **Removing stopwords**

**Extract the stop words**

In [9]:

*# extracting the stopwords from nltk library*

sw = stopwords.words('english')

*# displaying the stopwords*

np.array(sw)

Out[9]:

array(['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you',

'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his',

In [10]:

print("Number of stopwords: ", len(sw))

Number of stopwords: 153

**Function to remove stopwords**

In [11]:

def stopwords(text):

*'''a function for removing the stopword'''*

*# removing the stop words and lowercasing the selected words*

text = [word.lower() for word **in** text.split() if word.lower() **not** **in** sw]

*# joining the list of words with space separator*

return " ".join(text)

**Apply the function to each examples**

In [12]:

data['text'] = data['text'].apply(stopwords)

data.head(10)

Out[12]:

|  | id | text | author |
| --- | --- | --- | --- |
| 0 | id26305 | process however afforded means ascertaining di... | EAP |
| 1 | id17569 | never occurred fumbling might mere mistake | HPL |
| 2 | id11008 | left hand gold snuff box capered hill cutting ... | EAP |
| 3 | id27763 | lovely spring looked windsor terrace sixteen f... | MWS |
| 4 | id12958 | finding nothing else even gold superintendent ... | HPL |
| 5 | id22965 | youth passed solitude best years spent gentle ... | MWS |
| 6 | id09674 | astronomer perhaps point took refuge suggestio... | EAP |
| 7 | id13515 | surcingle hung ribands body | EAP |
| 8 | id19322 | knew could say stereotomy without brought thin... | EAP |
| 9 | id00912 | confess neither structure languages code gover... | MWS |

## **Top words before stemming**

**Collect vocabulary count**

We will not use word counts as feature for NLP since tf-idf is a better metric

In [13]:

*# create a count vectorizer object*

count\_vectorizer = CountVectorizer()

*# fit the count vectorizer using the text data*

count\_vectorizer.fit(data['text'])

*# collect the vocabulary items used in the vectorizer*

dictionary = count\_vectorizer.vocabulary\_.items()

Store the vocab and counts in a pandas dataframe

In [14]:

*# lists to store the vocab and counts*

vocab = []

count = []

*# iterate through each vocab and count append the value to designated lists*

for key, value **in** dictionary:

vocab.append(key)

count.append(value)

*# store the count in panadas dataframe with vocab as index*

vocab\_bef\_stem = pd.Series(count, index=vocab)

*# sort the dataframe*

vocab\_bef\_stem = vocab\_bef\_stem.sort\_values(ascending=False)

**Bar plot of top words before stemming**

In [15]:

top\_vacab = vocab\_bef\_stem.head(20)

top\_vacab.plot(kind = 'barh', figsize=(5,10), xlim= (25230, 25260))

Out[15]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f23c9e7d828>

## **Stemming operations**

Stemming operation bundles together words of same root. E.g. stem operation bundles "response" and "respond" into a common "respon"

**A funtion to carry out stemming operation**

In [16]:

*# create an object of stemming function*

stemmer = SnowballStemmer("english")

def stemming(text):

*'''a function which stems each word in the given text'''*

text = [stemmer.stem(word) for word **in** text.split()]

return " ".join(text)

**Apply the function to each examples**

In [17]:

data['text'] = data['text'].apply(stemming)

data.head(10)

Out[17]:

|  | id | text | author |
| --- | --- | --- | --- |
| 0 | id26305 | process howev afford mean ascertain dimens dun... | EAP |
| 1 | id17569 | never occur fumbl might mere mistak | HPL |
| 2 | id11008 | left hand gold snuff box caper hill cut manner... | EAP |
| 3 | id27763 | love spring look windsor terrac sixteen fertil... | MWS |
| 4 | id12958 | find noth els even gold superintend abandon at... | HPL |
| 5 | id22965 | youth pass solitud best year spent gentl femin... | MWS |
| 6 | id09674 | astronom perhap point took refug suggest non l... | EAP |
| 7 | id13515 | surcingl hung riband bodi | EAP |
| 8 | id19322 | knew could say stereotomi without brought thin... | EAP |
| 9 | id00912 | confess neither structur languag code govern p... | MWS |

## **Top words after stemming operation**

**Collect vocabulary count**

In [18]:

*# create the object of tfid vectorizer*

tfid\_vectorizer = TfidfVectorizer("english")

*# fit the vectorizer using the text data*

tfid\_vectorizer.fit(data['text'])

*# collect the vocabulary items used in the vectorizer*

dictionary = tfid\_vectorizer.vocabulary\_.items()

**Bar plot of top words after stemming**

In [19]:

*# lists to store the vocab and counts*

vocab = []

count = []

*# iterate through each vocab and count append the value to designated lists*

for key, value **in** dictionary:

vocab.append(key)

count.append(value)

*# store the count in panadas dataframe with vocab as index*

vocab\_after\_stem = pd.Series(count, index=vocab)

*# sort the dataframe*

vocab\_after\_stem = vocab\_after\_stem.sort\_values(ascending=False)

*# plot of the top vocab*

top\_vacab = vocab\_after\_stem.head(20)

top\_vacab.plot(kind = 'barh', figsize=(5,10), xlim= (15120, 15145))

Out[19]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f23c9965eb8>

## **Histogram of text length of each writer**

A function to return the length of text

In [20]:

def length(text):

*'''a function which returns the length of text'''*

return len(text)

Apply the function to each example

In [21]:

data['length'] = data['text'].apply(length)

data.head(10)

Out[21]:

|  | id | text | author | length |
| --- | --- | --- | --- | --- |
| 0 | id26305 | process howev afford mean ascertain dimens dun... | EAP | 136 |
| 1 | id17569 | never occur fumbl might mere mistak | HPL | 35 |
| 2 | id11008 | left hand gold snuff box caper hill cut manner... | EAP | 113 |
| 3 | id27763 | love spring look windsor terrac sixteen fertil... | MWS | 137 |
| 4 | id12958 | find noth els even gold superintend abandon at... | HPL | 102 |
| 5 | id22965 | youth pass solitud best year spent gentl femin... | MWS | 265 |
| 6 | id09674 | astronom perhap point took refug suggest non l... | EAP | 75 |
| 7 | id13515 | surcingl hung riband bodi | EAP | 25 |
| 8 | id19322 | knew could say stereotomi without brought thin... | EAP | 267 |
| 9 | id00912 | confess neither structur languag code govern p... | MWS | 80 |

**Extracting data of each class**

In [22]:

EAP\_data = data[data['author'] == 'EAP']

HPL\_data = data[data['author'] == 'HPL']

MWS\_data = data[data['author'] == 'MWS']

**Histogram of text lenght of each writer**

As we can see the distributions coincides so it better to leave out text length as a feature for predictive modelling

In [23]:

matplotlib.rcParams['figure.figsize'] = (12.0, 6.0)

bins = 500

plt.hist(EAP\_data['length'], alpha = 0.6, bins=bins, label='EAP')

plt.hist(HPL\_data['length'], alpha = 0.8, bins=bins, label='HPL')

plt.hist(MWS\_data['length'], alpha = 0.4, bins=bins, label='MWS')

plt.xlabel('length')

plt.ylabel('numbers')

plt.legend(loc='upper right')

plt.xlim(0,300)

plt.grid()

plt.show()

# ****Top words of each writer and their count****

## **Edgar Allan Poe**

In [24]:

*# create the object of tfid vectorizer*

EAP\_tfid\_vectorizer = TfidfVectorizer("english")

*# fit the vectorizer using the text data*

EAP\_tfid\_vectorizer.fit(EAP\_data['text'])

*# collect the vocabulary items used in the vectorizer*

EAP\_dictionary = EAP\_tfid\_vectorizer.vocabulary\_.items()

*# lists to store the vocab and counts*

vocab = []

count = []

*# iterate through each vocab and count append the value to designated lists*

for key, value **in** EAP\_dictionary:

vocab.append(key)

count.append(value)

*# store the count in panadas dataframe with vocab as index*

EAP\_vocab = pd.Series(count, index=vocab)

*# sort the dataframe*

EAP\_vocab = EAP\_vocab.sort\_values(ascending=False)

*# plot of the top vocab*

top\_vacab = EAP\_vocab.head(20)

top\_vacab.plot(kind = 'barh', figsize=(5,10), xlim= (9700, 9740))

Out[24]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f23c2022b38>

## **Mary Shelley**

In [25]:

*# create the object of tfid vectorizer*

HPL\_tfid\_vectorizer = TfidfVectorizer("english")

*# fit the vectorizer using the text data*

HPL\_tfid\_vectorizer.fit(HPL\_data['text'])

*# collect the vocabulary items used in the vectorizer*

HPL\_dictionary = HPL\_tfid\_vectorizer.vocabulary\_.items()

*# lists to store the vocab and counts*

vocab = []

count = []

*# iterate through each vocab and count append the value to designated lists*

for key, value **in** HPL\_dictionary:

vocab.append(key)

count.append(value)

*# store the count in panadas dataframe with vocab as index*

HPL\_vocab = pd.Series(count, index=vocab)

*# sort the dataframe*

HPL\_vocab = HPL\_vocab.sort\_values(ascending=False)

*# plot of the top vocab*

top\_vacab = HPL\_vocab.head(20)

top\_vacab.plot(kind = 'barh', figsize=(5,10), xlim= (9300, 9330))

Out[25]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f23c9911668>

## **HP Lovecraft**

In [26]:

*# create the object of tfid vectorizer*

MWS\_tfid\_vectorizer = TfidfVectorizer("english")

*# fit the vectorizer using the text data*

MWS\_tfid\_vectorizer.fit(MWS\_data['text'])

*# collect the vocabulary items used in the vectorizer*

MWS\_dictionary = MWS\_tfid\_vectorizer.vocabulary\_.items()

*# lists to store the vocab and counts*

vocab = []

count = []

*# iterate through each vocab and count append the value to designated list*

for key, value **in** MWS\_dictionary:

vocab.append(key)

count.append(value)

*# store the count in panadas dataframe and vocab as index*

MWS\_vocab = pd.Series(count, index=vocab)

*# sort the dataframe*

MWS\_vocab = MWS\_vocab.sort\_values(ascending=False)

*# plot of the top vocab*

top\_vacab = MWS\_vocab.head(20)

top\_vacab.plot(kind = 'barh', figsize=(5,10), xlim= (7010, 7040))

Out[26]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f23c255c278>

**As we can see the top words of each writer are cleary distinct and are in huge numbers. Word Count or TF-IDF of the can provide a good feature**

# ****TF-IDF Extraction****

tf-idf weight is product of two terms: the first term is the normalized Term Frequency (TF), aka. the number of times a word appears in a document, divided by the total number of words in that document; the second term is the Inverse Document Frequency (IDF), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).

IDF(t) = log\_e(Total number of documents / Number of documents with term t in it).

In [27]:

*# extract the tfid representation matrix of the text data*

tfid\_matrix = tfid\_vectorizer.transform(data['text'])

*# collect the tfid matrix in numpy array*

array = tfid\_matrix.todense()

In [28]:

*# store the tf-idf array into pandas dataframe*

df = pd.DataFrame(array)

df.head(10)

Out[28]:

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... | 15132 | 15133 | 15134 | 15135 | 15136 | 15137 | 15138 | 15139 | 15140 | 15141 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.266318 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 6 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

10 rows × 15142 columns

# ****Training Model****

We are going to train Naive Bayes Classifier. Naive Bayes Classifier is a good choice given we have a medium sized dataset, NB classifier scales well and also NB classifier has been historically used in NLP tasks. We will train Multinomial and Bernoulli NB classifier, since they almost always outperfrom Gaussian NB classifier in NLP tasks

Adding the output to the dataframe

In [29]:

df['output'] = data['author']

df['id'] = data['id']

df.head(10)

Out[29]:

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... | 15134 | 15135 | 15136 | 15137 | 15138 | 15139 | 15140 | 15141 | output | id |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | EAP | id26305 |
| 1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | HPL | id17569 |
| 2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | EAP | id11008 |
| 3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | MWS | id27763 |
| 4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.266318 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | HPL | id12958 |
| 5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | MWS | id22965 |
| 6 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | EAP | id09674 |
| 7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | EAP | id13515 |
| 8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | EAP | id19322 |
| 9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | MWS | id00912 |

10 rows × 15144 columns

Features and output of the models

In [30]:

features = df.columns.tolist()

output = 'output'

*# removing the output and the id from features*

features.remove(output)

features.remove('id')

**Import neccassary sklearn modules**

In [31]:

from sklearn.naive\_bayes import GaussianNB, BernoulliNB, MultinomialNB

from sklearn.neighbors import KNeighborsClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score, log\_loss

from sklearn.model\_selection import GridSearchCV

## **Tuning Multinomial Naive Bayes Classifier**

List of alpha parameter we are going to try

In [32]:

alpha\_list1 = np.linspace(0.006, 0.1, 20)

alpha\_list1 = np.around(alpha\_list1, decimals=4)

alpha\_list1

Out[32]:

array([ 0.006 , 0.0109, 0.0159, 0.0208, 0.0258, 0.0307, 0.0357,

0.0406, 0.0456, 0.0505, 0.0555, 0.0604, 0.0654, 0.0703,

0.0753, 0.0802, 0.0852, 0.0901, 0.0951, 0.1 ])

GridSearchCV allows us tune parameters of a model through k-fold cross validataion using parameter grid in one go

**Gridsearch**

In [33]:

*# parameter grid*

parameter\_grid = [{"alpha":alpha\_list1}]

In [34]:

*# classifier object*

classifier1 = MultinomialNB()

*# gridsearch object using 4 fold cross validation and neg\_log\_loss as scoring paramter*

gridsearch1 = GridSearchCV(classifier1,parameter\_grid, scoring = 'neg\_log\_loss', cv = 4)

*# fit the gridsearch*

gridsearch1.fit(df[features], df[output])

Out[34]:

GridSearchCV(cv=4, error\_score='raise',

estimator=MultinomialNB(alpha=1.0, class\_prior=None, fit\_prior=True),

fit\_params=None, iid=True, n\_jobs=1,

param\_grid=[{'alpha': array([ 0.006 , 0.0109, 0.0159, 0.0208, 0.0258, 0.0307, 0.0357,

0.0406, 0.0456, 0.0505, 0.0555, 0.0604, 0.0654, 0.0703,

0.0753, 0.0802, 0.0852, 0.0901, 0.0951, 0.1 ])}],

pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score='warn',

scoring='neg\_log\_loss', verbose=0)

Collect results in pandas dataframe

In [35]:

results1 = pd.DataFrame()

*# collect alpha list*

results1['alpha'] = gridsearch1.cv\_results\_['param\_alpha'].data

*# collect test scores*

results1['neglogloss'] = gridsearch1.cv\_results\_['mean\_test\_score'].data

**Plot of logloss vs alpha**

In [36]:

matplotlib.rcParams['figure.figsize'] = (12.0, 6.0)

plt.plot(results1['alpha'], -results1['neglogloss'])

plt.xlabel('alpha')

plt.ylabel('logloss')

plt.grid()

In [37]:

print("Best parameter: ",gridsearch1.best\_params\_)

Best parameter: {'alpha': 0.020799999999999999}

In [38]:

print("Best score: ",gridsearch1.best\_score\_)

Best score: -0.441224371424

## **Tuning Multinomial Naive Bayes Classifier**

List of alpha parameter we are going to try

In [39]:

alpha\_list2 = np.linspace(0.006, 0.1, 20)

alpha\_list2 = np.around(alpha\_list2, decimals=4)

alpha\_list2

Out[39]:

array([ 0.006 , 0.0109, 0.0159, 0.0208, 0.0258, 0.0307, 0.0357,

0.0406, 0.0456, 0.0505, 0.0555, 0.0604, 0.0654, 0.0703,

0.0753, 0.0802, 0.0852, 0.0901, 0.0951, 0.1 ])

Parameter grid

In [40]:

parameter\_grid = [{"alpha":alpha\_list2}]

**Gridsearch**

In [41]:

*# classifier object*

classifier2 = MultinomialNB()

*# gridsearch object using 4 fold cross validation and neg\_log\_loss as scoring paramter*

gridsearch2 = GridSearchCV(classifier2,parameter\_grid, scoring = 'neg\_log\_loss', cv = 4)

*# fit the gridsearch*

gridsearch2.fit(df[features], df[output])

Out[41]:

GridSearchCV(cv=4, error\_score='raise',

estimator=MultinomialNB(alpha=1.0, class\_prior=None, fit\_prior=True),

fit\_params=None, iid=True, n\_jobs=1,

param\_grid=[{'alpha': array([ 0.006 , 0.0109, 0.0159, 0.0208, 0.0258, 0.0307, 0.0357,

0.0406, 0.0456, 0.0505, 0.0555, 0.0604, 0.0654, 0.0703,

0.0753, 0.0802, 0.0852, 0.0901, 0.0951, 0.1 ])}],

pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score='warn',

scoring='neg\_log\_loss', verbose=0)

Collect results in pandas dataframe

In [42]:

results2 = pd.DataFrame()

*# collect alpha list*

results2['alpha'] = gridsearch2.cv\_results\_['param\_alpha'].data

*# collect test scores*

results2['neglogloss'] = gridsearch2.cv\_results\_['mean\_test\_score'].data

**Plot of logloss vs alpha**

In [43]:

matplotlib.rcParams['figure.figsize'] = (12.0, 6.0)

plt.plot(results2['alpha'], -results2['neglogloss'])

plt.xlabel('alpha')

plt.ylabel('logloss')

plt.grid()

In [44]:

print("Best parameter: ",gridsearch2.best\_params\_)

Best parameter: {'alpha': 0.020799999999999999}

In [45]:

print("Best score: ",gridsearch2.best\_score\_)

Best score: -0.441224371424