**Natural Language Processing using Python**

===================================================================

In this tutorial you will learn how to implement *basics of natural language processing using python*. You will learn about text processing and some of the very important asspects of Natural Language Processing (NLP) viz. **tokenization, cleaning data, frequency distribution plots, Dispersion plots** apart from the basic **searching, finding similar words** kind of tasks. You will learn how to work with ***NLTK*** library.

You will also learn on getting features from the text...

*So...Here you GO....*

**Introduction to NLP**

In reality, plain text is the most predominant form of data available today. Text analysis applies analysis of word frequency distributions, pattern recognition, tagging, link and association analysis, sentiment analysis, and visualization. Natural Language Processing is a broad topic, Machine Translation, Summarizing texts, spam detection, sentiment analysis are real big field. Python is readable, fast for prototypes, it has rich library for reading and writing data, running calculations on the information and creating graphical representations of data sets and list support, it includes a lot of NLP-related libraries viz. **NLTK, textblob, scipy, pandas**... also it has great parsing libraries viz. **Beautifulsoup, scrapy**.

Natural Language Toolkit (NLTK) is a Python API for the analysis of texts written in natural languages, such as English. NLTK is a very popular and old library which comes with a collection of sample texts called corpora (collection of text documents).

NLTK is a leading platform for building Python programs to work with human language data (Natural Language Processing).

NLTK is intended to support research and teaching in NLP or closely related areas, including empirical linguistics, cognitive science, artificial intelligence, information retrieval, and machine learning.

**Text Preprocessing**

**Accessing data:-**

Well you need to have an access to your data before starting to process it. You can access and open your data file in many ways.

* #### Opening a local file

When you have the data file on your local machine, you can access you file using ***open()*** method and can read into the lines using ***.readline()***. Don't forget to mention **'r'** mode while opening a file.

f=open('text.txt', 'r')

f.readline()

* #### Opening file from other online source You might have to process data/text which is available online, then you can fetch the required data like this:-
* **import** **requests**
* f\_online='http://www.gutenberg.org/files/11111/11111.txt'
* f\_rawtext=requests.get(f\_online).text
* **print** (f\_rawtext)
* #### Opening file from corpus As mentioned earlier that nltk comes with a collection of sample texts called corpora (collection of text documents). You can access text files from this corpus as well. A complete list is available at <http://www.nltk.org/nltk_data/>.

Visit <http://www.nltk.org/install.html> to ***install nltk*** on your machine.

You can install **nltk data** as follows:-

**import** **nltk**

nltk.download()

Once you execute the above line of codes, a new window will appear which is a *NLTK Downloader*. You can download the entire collection by using **all**, or just the data required for your project/assignment. You can also download *Gutenberg* from this link <http://www.nltk.org/nltk_data/> .

Here you can open *Gutenberg* from **nltk.corpus** and create sample text from **'bible-kjv.txt'** file from *Gutenberg*.

**from** **nltk.corpus** **import** gutenberg

textguten = gutenberg.raw("bible-kjv.txt")

*Note* that **nltk.corpus** needs to be downloaded beforehand if you want to work on corpus data.

**Text searching:-**

Before getting into the ways of searching text from text files let's import some text.

**from** **nltk.corpus** **import** gutenberg

textguten = gutenberg.raw("bible-kjv.txt")

You can see that object textguten carries a string. You cannot apply 'Text' funstions on a string. You need to import **Text** module from *nltk.text* and you need to fetch words from gutenberg first than raw string.

**from** **nltk.text** **import** Text

**from** **nltk.corpus** **import** gutenberg

textguten=gutenberg.words("bible-kjv.txt")

text=Text(textguten)

*However if you run this code:-*

**from** **nltk.book** **import** \*

*You will find that the output displays all the****text****files available in the nltk with the index as text1, text2, text 3 and so on till text9 (as it has 9 text files) along with the title and the author, if available, details as sentence (sent1, sent2...sent9). You can directly call text functions like concordance, similar on these text files. You can simple get to know about the text by typing in the command line the text name as sown below:-*

text7

*So you can see that****tex7****is a journal named Wall street journal.*

You have text files now, you can try some text searching commands on it. You can start with 'concordance'. The syntax to use concordance on a text file is as:-

***syntax:-* text\_file\_name.concordance("word\_to\_be\_searched")**

**You have fetched text from Gutenberg and converted that string into *Text*. This text is stored in the variable 'text'. You can now try text searching on this.**

Concordance produces all occurrences of a specified word along some context. Using concordance you can search for the incidents where word 'foundation' appears in the text file text1.Concordance allows you to spot lexis (words) in context.

text.concordance("God")

*Note that it also specifies the total number of occurrences of the specified word ['Displaying 25 of 4472 matches' as for your word 'God'] at the start.*

What if you want to search for words similar to a given word in the text?

**syntax:- text\_file\_name.similar("word\_to\_be\_searched")**

You can try finding similar words of a given word in your *'text'*.

text.similar("God")

text.similar("monster")

text.similar("comit")

text.similar("said")

The command .similar() displays words which appear in a similar range of contexts. See output for commands, where similar("monster") and similr("comit").... *no match is found!*.

\*Note that we get different results for different texts even for the same word.\*

What if you want to look for the contexts which are common for two words? .common\_contexts() allows you to do so.

**syntax:- name\_of\_the\_file.common\_contexts(["word1", "word1"])**

text.common\_contexts(["God", "thee"])

Usage of collocations is frequent in business and work settings. A collocation is a series of words that happen to appear together often oddly. Like blue sky, feel free, close a deal are collocations. This is how you can look for collocations inside the text:-

**syntax:- name\_of\_the\_file.collocations()**

text.collocations()

Collocation helps you find **bigrams** that occur more often than you would expect based on the frequency of the individual words

**Tokenization:-**

Tokenizing means splitting your document into lexical chunks.

Tokenizers are used to **break text strings** and to *find words and punctuation* in a string.

Tokenizing can be done at *word level* and at *sentence level*. NLTK has a few tokenizers packages. The following sample code requires Punkt sentence tokenization models to be installed.

You need a paragraph kind of text to undersand what tokenization does. Well, at the start of this tutorial, you fetched raw data from the same *Gutenberg*. You can make use of that **textguten**!

textguten= gutenberg.raw("bible-kjv.txt")

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

**print**(sent\_tokenize(textguten[:1000]))*#considering only first 1000 alphabets from textguten*

**print**(word\_tokenize(textguten[:1000]))

**Stop Words:-**

These are common terms which rarely add to the meaning of a sentence for the purposes of NLP and information retrieval. These words are recurring many a times as part of speech and need not be considered for classification.

In textguten[:1000], you can see, that you get following tokens:-

*['****[****', '****The****', 'King', 'James', 'Bible', ']', 'The', 'Old', 'Testament', 'of', 'the', 'King', 'James', 'Bible', 'The', 'First', 'Book', '****of****', 'Moses', ':', 'Called', 'Genesis', '****1:1****', 'In', 'the', 'beginning', 'God', 'created', 'the', 'heaven', 'and', 'the', 'earth', '.', '****1:2****', 'And', 'the', 'earth', 'was', 'without', 'form', ',', 'and', 'void', ';', '****and****', 'darkness', 'was', 'upon', 'the', 'face', 'of', 'the', 'deep', '****.****', 'And', 'the', 'Spirit', 'of', 'God', 'moved', 'upon', 'the', 'face', 'of', 'the', 'waters', '.', '****1:3****', 'And', 'God', 'said', '****,****', 'Let', 'there', 'be', 'light', ':', 'and', 'there', 'was', 'light', '.', '****1:4****', 'And', 'God', 'saw', 'the', 'light', ',', 'that', 'it', 'was', 'good', ':', 'and', 'God', 'divided', 'the', 'light', 'from', 'the', 'darkness', '.', '1:5', 'And', 'God', 'called', 'the', 'light', 'Day', ',', 'and', 'the', 'darkness', 'he', 'called', 'Night', '.', 'And', 'the', 'evening', 'and', 'the', 'morning', 'were', 'the', 'first', 'day', '.', '1:6', 'And', 'God', 'said', ',', 'Let', 'there', 'be', 'a', 'firmament', 'in', 'the', 'midst', 'of', 'the', 'waters', ',', 'and', 'let', 'it', 'divide', 'the', 'waters', 'from', 'the', 'waters', '.', '1:7', 'And', 'God', 'made', 'the', 'firmament', ',', 'and', 'divided', 'the', 'waters', 'which', 'were', 'under', 'the', 'firmament', 'from', 'the', 'waters', 'which', 'were', 'above', 'the', 'firmament', ':', 'and', 'it', 'was', 'so', '.', '1:8', 'And', 'God', 'called', 'the', 'firmament', 'Heaven', '.', 'And', 'the', 'evening', 'and', 'the', 'morning', 'were', 'the', 'second', 'day', '.', '1:9', 'And', 'God', 'said', ',', 'Let', 'the', 'waters', 'under', 'the', 'heav']*

you can create a list of stopwords (like the ones which are ***bold***) from the above tokens, as follows:- You can eliminate stop words from the *textguten* either using **list comprehensioon** or by implementing **for loop**.

stop=['[',']','The','of','1:1',':','.','1:2',',','1:3',':','1:4','1:5','1:6','it','1:7','In','1:9','1:8','so','Let']

words = word\_tokenize(textguten[:1000])

clean\_textguten= [i **for** i **in** words **if** **not** i **in** stop]*#using list comprehension*

*#using for loop*

clean\_textguten = []

**for** i **in** words:

**if** i **not** **in** stop:

clean\_textguten.append(i)

**print**(clean\_textguten)

After this, you have got clean text (*clean\_textguten*)!!

See the output does not contain the tokens which were present in the list of stopwords.

*Try the following code!* Another way of creating stopwords is by using stopwords module from nltk.corpus as follows:-

**from** **nltk.corpus** **import** stopwords

stop = set(stopwords.words('English'))

*Here, you are creating a set of 'English' stop words. Well if you are working on say French text, you can create French stopwords as well.*

**Lexicon Normalization:-**

To perform normalization that mainly involves eliminating punctuation, converting the entire text into lowercase or uppercase, converting numbers into words, expanding abbreviations, canonicalization of text, and so on. As text is fetched in the form of a string, .lower() and .upper() can be applied on it for converting text into lowercase or uppercase. The commonly used normalization methods are stemming and lemmatization.

**Stemming:-**

Words need to be stemmed to retrieve their radicals, so that various forms derived from a stem would be taken as the same when counting word frequency. Stemming is used to remove morphological affixes from words, leaving only the word stem. For instance, words “update", “updated" and “updating" would all be stemmed to “update".

For stemming, you need to import some stemmer from *nltk.stem module*. Then create an object for that stemmer. And then you can find the stems for all the words in the *clean text* using for loop.

**from** **nltk.stem** **import** PorterStemmer

ps = PorterStemmer()

stemtext=[]

**for** i **in** clean\_textguten:

p=ps.stem(i)

stemtext.append(p)

**print**(stemtext)

**Lemmatization:-**

This operation is similar to stemming. Let's lemmatize a few words. First you need to import a lemmatizer. You can use WordNetLemmatizer from nltk.stem package. Then create an instance of that lemmatizer and then use .lemmatize() for getting the stems of words, as follows:-

</dl>

**from** **nltk.stem** **import** WordNetLemmatizer

lr = WordNetLemmatizer()

lemtext=[]

**for** i **in** clean\_textguten:

p=lr.lemmatize(i)

lemtext.append(p)

**print**(lemtext)

Here, you are importing WordNetLemmatizer, then **lr** object is created for this lemmatizer. And you are creating a list of lemmatized word (*lemtext*) for the words present in \_clean*textguten*.

* Note that it returns the input word unchanged if it cannot be found in WordNet.

**Text to Features (Feature Engineering on text data)**

**Part of Speech Tagging:-**

This process converts tokenizes words into a list of containing word and its corresponding POS. A part-of-speech tag signifies whether the word is a noun, adjective, verb, and so on.

You can start with tagging POS to your text!

First of all, you need to tokenize the string in to words using word\_tokenize, (*if required*) and then you need create a list called 'words' of the tokens so created.

Using pos\_tag from nltk, you can tag tokens to their respective POS. Well, Later you can print those tags.

By now, you have your clean text (clean\_textguten), which is already in the form of tokens, so you will not require to tokenize it!

tagged\_textguten = nltk.pos\_tag(clean\_textguten)

**print**(tagged\_textguten[:2])*#printing results for only first two instances!*

It is clear from the output that 'the' is determiner as indicated by DT and 'Project' is 'Noun, Proper, Singular' as indicated by NNP. Note that tagger creates a list of tuple containing word and its corresponding POS.

**POS tagging is an essential step prior to making chunks as chunker cannot recognize extract phrases from a sentence without the part-of-speech tags.**

**Chunking:-**

Chunks are the minimal units that can be processed or we can say that chunks are the non-overlapping linguistic groups. Chunking is a kind of shallow parsing. Chunking is not possible to do until we have done tagging.

</dl> You need to define a chunk grammar i.e. You need to create a rule (regular expression based on tags) for the entity to be extracted. To do this, you're going to use regular expressions. A regular expression is a set of characters used to define a search pattern. Let's say you want to search for words begning with 'a', you need to create a regex which will be 'a\w+' in your case. Where 'a' at the start of regex means will only match words that begin with 'a'; '\w' is a special character and it will match any alphanumeric A-z, a-z, 0-9, along with underscores;'+' means the regex can appear many times in strings.

for example:- grammar = "NP: {\<DT>\?

* = match 1 or more ? = match 0 or 1 repetitions.
* = match 0 or MORE repetitions  
  . = Any character except a new line

**for** i **in** clean\_textguten[:2]:

chunkGram = r"""Chunk: {<RB.?>\*<VB.?>\*<NNP>+<NN>?}"""

chunkParser = nltk.RegexpParser(chunkGram)

chunked = chunkParser.parse(tagged)

chunked.draw()

**print**(chunked)

Note that the chunkgram n the above example code means:- <RB.?>*= \"0 or more of any tense of adverb, <VB.?>* = \"0 or more of any tense of verb,

**Named Entity Recognition:-**

Named entity is a real-world object, such as persons, locations, organizations, products, etc., that can be denoted with a proper name (https://en.wikipedia.org/wiki/Named\_entity). Named entity recognition is a way of extracting most common entities such as people, places, things, locations, monetary figures, and more. With named entity recognition you can easily locate proper names of people, organizations, locations, or other entities from the text. After tagging you can use \_\_nltk.ne\_chunk()\_\_, nltk provides a classifier trained to recognize named entities, accessed with the function nltk.ne\_chunk().

</dl>

**for** i **in** clean\_text[:2]:

ner = nltk.ne\_chunk(tagged, binary=True)

ner.draw()

Note that if you set binary=True, then named entities are just tagged as NE; otherwise, the classifier adds category labels such as Person, Organization, and GPE.

**for** i **in** clean\_text[:2]:

ner = nltk.ne\_chunk(tagged)

ner.draw()

**TF – IDF :-**

A term-document matrix represents the relationship between terms and documents, where each row stands for a term or words as present in the data sheet and each column for a document, and an entry is the number of occurrences of the term in the document.

</dl> Given a word, a document and a collection of documents, Vocabulary of the document is the number of unique/different words that document contains. Or we can say that The vocabulary of a text is just the set of tokens that it uses.

Term Frequency is the count of words in the document **(vocab) / length of the document**.

Inverse Document Frequency is the **log of length of collection / count of document\_containing\_term in the collection**.

**Term Frequency – Inverse Document Frequency = tf \* idf**

Note nltk does not have anything to find *Term Frequency – Inverse Document Frequency* directly. However you can do so using **len()** method. This is explained under the head of Lexical richness.

**Lexical richness:-**

Lexical richness means number of the number of distinct words in the document.

Function for finding lexical richness of the text:-

**def** lexical\_richness(text):

**return** len(set(text)) / len(text)

Function for finding percentage of the text is taken up by a specific word:-

**def** percentage(word, text):

**return** (100 \* text.count(word) / len(text))

Now you can call both functions for your text *textguten*

**print** ('Lexical richness of the text: '+str(lexical\_richness(textguten)))

**print** ('Percentage: '+ str(percentage('only',(textguten))))

*Using****math.log()****, you can easily find the IDF*

**Frequency Distribution:-**

Frequency distribution notify about the frequency of each vocabulary item in the text.

Using following code you can find the frequency distribution of *textguten*. You need to import *FreqDist* beforehand from *nltk*.

**from** **nltk** **import** FreqDist

fdist\_textguten = FreqDist(textguten)

Note that output is a *dictionary* containing every *vocabulary item* with the value of its *occurrence*. You can also display this dictionary using *print()* command.

~python

print(fdist\_textguten)

You can also find most common 10 vocab items of this text.

fdist\_textguten.most\_common(10)

Well, you can also plot frequency distribution graph. If you wish to plot frequency distribution, try following code, which plots frequency distribution of 10 most common vocab items.

fdist\_textguten.plot(10, cumulative=True)

**Dispersion plot:-**

Dispersion plots determines the location of a word in the text: how many words from the beginning it appears.

With dispersion plot you can find the positional information of a word.

**"syntax:- name\_of\_the\_file.dispersion\_plot([\"word1\", \"word2\", \"word3\"...])"**

textguten=gutenberg.words("bible-kjv.txt")

text=Text(textguten)*#dispersion\_plot does not work on strings!*

text.dispersion\_plot(["God", "James", "Bible"])

text.dispersion\_plot(['Testament', 'created'])

With the help of these plots you can easily visualize that how many times a word has appeared in your text (*textguten*).

**Finding synonym:-**

WordNet is a lexical database for the English language and can be used to find the meanings of words, synonyms, antonyms, and more.

</dl> For finding synonyms of a word, you first need to import **wordnet** from nltk and then you can call **syssets()** on the word for which you wish to know the synomym.

Well to fins a synonym you need to have a word, for which you can find it's synonym! **clean\_textguten** contains words only from textguten. You can make use of it here or you can tokenize textguten into words for finding synonyms.

*#word\_textguten=word\_tokenize(textguten[:100])#use for tokenizing!*

**from** **nltk.corpus** **import** wordnet

syns=[]

**for** i **in** clean\_textguten[:2]:*#considering only two words!*

syns.append( wordnet.synsets(i))

**print**(syns)

The output for the above command will be a list of synonyms for each word. *You can easily look inside this list of synonyms using indexing as follows:- print(syns[0].name()), print(syns[0].definition()), print(syns[0].examples())*

* Creating list of synonyms and antonyms of a word by using **.lemmas()** and **.antonyms()**.

synonyms = []

antonyms = []

**for** i **in** clean\_textguten[:2]:

s=wordnet.synsets(i)

**for** syn **in** s:

**for** l **in** syn.lemmas():

synonyms.append(l.name())

**if** l.antonyms():

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

**print**(set(synonyms))

**print**(set(antonyms))

The coding skills you have learned are going to be tremondously helpful and will be relevant to a significant part of data, called Unstructured Data and incorporates a lot of content.

Thanks for reading, and I hope to see you in the next tutorial!, where you will learn to classify unstructured data using models like Naive Bayes.

In [ ]: