Python Data Science Getting Started Tutorial: NLTK

SWAYAM MITTAL Follow
Feb 11 · 56 min read

Photo by Fotis Fotopoulos on Unsplash

I. Analyze words and sentences using NLTK

Welcome to the Natural Language Processing series of tutorials, using Python's natural language toolkit NLTK module.

The NLTK module is a huge toolkit designed to help you with the entire Natural Language Processing (NLP) approach. NLTK will provide you with everything from splitting paragraphs to sentences, splitting words, identifying the part of speech, highlighting themes, and even helping your machine understand what the text is about. In this series, we will address the areas of opinion mining or sentiment analysis.

As we learn how to use NLTK for sentiment analysis, we will learn the following:

- Participle—Splits the text body into sentences and words.
- Part of speech tagging
- Machine learning and naive Bayes classifier
- How to use Scikit Learn (sklearn) with NLTK together
- Training classifiers with data sets
- Real-time streaming sentiment analysis with Twitter.
- …and more.

To get started, you need the NLTK module, as well as Python.

If you don't already have Python, go to `python.org` and download the latest version of Python (if you are on Windows). If you are on Mac or Linux, you should be able to run `apt-get install python3` .

Next, you need NLTK 3. The easiest way to install an NLTK module is to use `pip` .

For all users, this is done by opening `cmd.exe` , bash, or any shell you use and typing the following command:

```
Pip install nltk
```

Next, we need to install some components for NLTK. Open python in any of your usual ways and type:

```
Import nltk
Nltk.download()
```

Select Download All for all packages and click Download. This will give you all the word breakers, blockers, other algorithms and all the corpora. If space is an issue, you can choose to manually download all content. The NLTK module will take up approximately 7MB and the entire `nltk_data` directory will occupy approximately `nltk_data` , including your `nltk_data` , parser and corpus.

If you are running a headless version with VPS, you can install everything by running Python and doing the following:

```
Import nltk
Nltk.download()
d (for download)
All (for download everything)
```

This will download everything for you.

Now that you have all the things you need, let's type some simple words:

- Corpus—The body of the text, singular. Corpora is its plural. Example: `A collection of medical journals` .
- Lexicon—vocabulary and its meaning. For example: English dictionary. However, considering that there are different thesaurus in each field. For example, for financial investors, the first meaning of the word `Bull` is the person who is confident in the market. Compared with the "common English vocabulary", the first meaning of the word is animal. Therefore, financial investors, doctors, children, mechanics, etc. all have a special vocabulary.
- Token—Each "entity" is part of a rule based split. For example, when a sentence is "split" into words, each word is a tag. If you split a paragraph into sentences, each sentence can also be a marker.

These are the most frequently heard words when entering the natural language processing (NLP) field, but we will cover more words in time. So, let's show an example of how to split something into tags with the NLTK module.

```
From nltk.tokenize import sent_tokenize, word_tokenize

EXAMPLE_TEXT = "Hello Mr. Smith, how are you doing today?
The weather is great, and Python is awesome. The sky is
pinkish-blue. You shouldn't eat cardboard."

Print(sent_tokenize(EXAMPLE_TEXT))
```

At first, you might think that word segmentation by word or sentence is quite a trivial matter. For many sentences, it may be. The first step might be to execute a simple `.split('. ')` , or by a period, followed by a space split. Then maybe you will introduce some regular expressions, divided by periods, spaces, and then uppercase letters. The problem is something like `Mr. Smith` , and there are many other things that will cause you trouble. Segmentation by word is also a challenge, especially when considering abbreviations, such as `we` and `we're` . NLTK saves you a lot of time with this seemingly simple but very complicated operation.

The above code will output the sentence, divided into a list of sentences, you can use the `for` loop to traverse.

```
[ 'Hello Mr. Smith, how are you doing today?', 'The weather
is great, and Python is awesome.', 'The sky is pinkish-
blue.', "You shouldn't eat cardboard." ]
```

So here we create the tags, they are all sentences. Let us divide the words by word this time.

```
Print (word_tokenize(EXAMPLE_TEXT))

[ 'Hello' , 'Mr.' , 'Smith' , ',' , 'how' , 'are' , 'you '
, 'doing ' , 'today ' , '?' , 'The' , 'weather' , 'is' ,
'great' , ',' , 'and' , 'Python' , 'is' , 'awesome ' , '.'
, 'The' , 'sky' , 'is' , 'pinkish-blue' , '.' , 'You' ,
'should' , "n't" , 'eat' , 'cardboard' , '.' ]
```

There are a few things to note here. First, notice that punctuation is treated as a separate tag. Also, note that the word `shouldn't` separated into `should` and `n't` . The last thing to note is that `pinkish-blue` is indeed treated as "a word", which is what it is. Cool!

Now, looking at the words after these participles, we must start thinking about what our next step might be. We began to think about how to get the meaning by observing these words. We can think about how to put value on many words, but we also see some words that are basically worthless. This is a form of "stop word" that we can also handle. This is what we will discuss in the next tutorial.

II. NLTK and stop words

The idea of natural language processing is to perform some form of analysis or processing. The machine can at least understand the meaning, representation or suggestion of the text to some extent.

This is obviously a huge challenge, but there are steps that anyone can follow. However, the main idea is that computers don't understand words directly. It is shocking that humans will not. In humans, memory is broken down into electrical signals in the brain in the form of a neural group that emits patterns. There are still many unknown things about the brain, but the more we break down the human brain into basic elements, we will find the basic elements. Well, it turns out that computers store information in a very similar way! If we want to mimic how humans read and understand text, we need a way that is as close as possible. In general, computers use numbers to represent everything, but we often see the use of binary signals directly in programming (`True` or `False` , which can be directly converted to 1 or 0, directly from the presence of an electrical signal `(True, 1)` or not Exist `(False, 0)`). To do this, we need a way to convert words to numeric or signal patterns. Converting data into something that a computer can understand is called "preprocessing." One of the main forms of preprocessing is to filter out useless data. In natural language processing, useless words (data) are called stop words.

We can immediately realize that some words are more meaningful than others. We can also see that some words are useless and are filled words. For example, we use them in English to fill sentences, so there is no such strange sound. One of the most common, unofficial, useless examples is the word `umm` . People often use `umm` to fill, more than other words. This word is meaningless unless we are looking for someone who may lack confidence, confusion, or not much. We all do this, there are… oh… many times, you can hear me say `umm` or `uhh` in the video. For most analyses, these words are useless.

We don't want these words to take up space in our database or take up valuable processing time. Therefore, we call these words "useless words" because they are useless and we want to treat them. Another version of the word "stop word" can be written more: the words we stop at.

For example, if you find words that are often used for satire, you may want to stop immediately. Satirical words or phrases will vary depending on the thesaurus and corpus. For the time being, we will treat stop words as words that do not contain any meaning, and we will remove them.

You can do it easily by storing a list of words that you think are stop words. NLTK uses a bunch of words that they think are stop words to get you started, you can access it through the NLTK corpus:

```
From nltk.corpus import stopwords
```

Here is the list:

```
>>> 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's how to use the `stop_words` collection to remove stop words from the 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 is:

```
['This', 'is', 'a', 'sample', 'sentence', ',', 'showing',
'off', 'the', 'stop', 'words', 'filtration', ' .']
['This', 'sample', 'sentence', ',', 'showing', 'stop',
'words', 'filtration', '.']
```

Thanks to our database. Another form of data preprocessing is "Stemming," which is what we'll discuss next.

III. NLTK stem extraction

The concept of stemming is a standardized approach. In addition to the tense, many variations of the word have the same meaning.

The reason we extract the stem is to shorten the time of the search and normalize the sentence.

consider:

```
I was taking a ride in the car.
I was riding in the car.
```

These two sentences mean the same thing. `in the car` (in the car) is the same. `I` (I) is the same. In both cases, `ing` clearly expresses the past tense, so in the case of trying to figure out the meaning of this past-style activity, is it really necessary to distinguish between `riding` and `taking a ride` ?

No, not.

This is just a small example, but imagine every word in English that can be placed on every possible tense and affix on the word. Each version has a separate dictionary entry that will be very redundant and inefficient, especially since once we convert to numbers, the "value" will be the same.

One of the most popular porcelain extraction algorithms is Porter, which existed in 1979.

First, we have to crawl and define our stems:

```
From nltk.stem import PorterStemmer
From nltk.tokenize import sent_tokenize, word_tokenize

Ps = PorterStemmer()
```

Now let's choose some words with similar stems, for example:

```
Example_words = [ "python" , "pythoner" , "pythoning" ,
"pythoned" , "pythonly" ]
```

Below, we can do this to easily extract stems:

```
For w in example_words:
   Print (ps.stem(w))
```

Our output:

```
Python
Python
Python
Python
Pythonli
```

Now let's try to extract stems from a typical sentence instead of some words:

```
New_text = "It is important to by very pythonly while you
are pythoning with python. All pythoners have pythoned
poorly at least once."
Words = word_tokenize(new_text)

For w in words :
    Print(ps.stem(w))
```

Our results now are:

```
It
Is
Import
To
By
Veri
Pythonli
While
You
Are
Python
With
Python
.
All
Python
Have
Python
Poorli
At
Least
Onc
.
```

Next, we'll discuss some of the more advanced content of the NLTK module, part-of-speech tagging, where we can use the NLTK module to identify the part of speech of each word in the sentence.

IV. NLTK part-of-speech tagging

A more powerful aspect of the NLTK module is that it can be used for your part-of-speech tagging. It means to mark words in a sentence as nouns, adjectives, verbs, etc. Even more impressive is that it can also be marked by tense, and others. This is a list of tags, their meaning and some examples:

```
POS tag list:
 CC coordinating conjunction
 CD cardinal digit
 DT determiner
 EX existential there ( like : "there is" ... think of it
like "there exists" )
 FW foreign word
 IN preposition/subordinating conjunction
 JJ adjective 'big'
 JJR adjective, comparative 'bigger'
 JJS adjective, superlative 'biggest'
 LS list marker 1 )
 MD modal could, will
 NN noun, singular 'desk'
 NNS noun plural 'desks'
 NNP proper noun, singular 'Harrison'
 NNPS proper noun, plural 'Americans'
 PDT predeterminer 'all the kids'
 POS possessive ending parent 's
 PRP personal pronoun I, he, she
 PRP$ possessive pronoun my, his, hers
 RB adverb very, silently,
 RBR adverb, comparative better
 RBS adverb, superlative best
 RP particle give up
 TO to go 'to' the store.
 UH interjection errrrrrrrm
 VB verb, base form take
 VBD verb, past tense took
 VBG verb, gerund/present participle taking
 VBN verb, past participle taken
 VBP verb, sing. present, non- 3 d take
 VBZ verb, 3 rd person sing. present takes
 WDT wh-determiner which
 WP wh-pronoun who, what
 WP$ possessive wh-pronoun whose
 WRB wh-abverb where , when
```

How do we use this? When we deal with it, we have to explain a new sentence marker called `PunktSentenceTokenizer` . This marker enables unsupervised machine learning, so you can actually train on any text you use. First, let's get some imports that we plan to use:

```
Import nltk
From nltk.corpus import state_union
From nltk.tokenize import PunktSentenceTokenizer
```

Now let’s create the training and test data:

```
Train_text = state_union.raw( "2005-GWBush.txt" )
Sample_text = state_union.raw( "2006-GWBush.txt" )
```

One is the State of the Union speech since 2005, and the other is the speech of President George W. Bush since 2006.

Next, we can train the Punkt marker as follows:

```
Custom_sent_tokenizer = PunktSentenceTokenizer(train_text)
```

After that we can actually divide the word and use:

```
Tokenized = custom_sent_tokenizer.tokenize(sample_text)
```

Now we can complete the part-of-speech tagging script by creating a function that will traverse and mark the part of speech of each sentence as follows:

```
Def process_content () :
   Try :
      For i in tokenized[: 5 ]:
         Words = nltk.word_tokenize(i)
         Tagged = nltk.pos_tag(words)
         Print(tagged)


   Except Exception as e:
        Print(str(e))



Process_content()
```

The output should be a tuple list, with the first element in the tuple being a word and the second element being a part of speech tag. It should look like:

```
[('PRESIDENT', 'NNP'), ('GEORGE', 'NNP'), ('W.', 'NNP'),
('BUSH', 'NNP'), ( "'S" , 'POS '), ('ADDRESS', 'NNP'),
('BEFORE', 'NNP'), ('A', 'NNP'), ('JOINT', 'NNP'),
('SESSION', 'NNP '), ('OF', 'NNP'), ('THE', 'NNP'),
('CONGRESS', 'NNP'), ('ON', 'NNP'), ('THE', 'NNP '),
('STATE', 'NNP'), ('OF', 'NNP'), ('THE', 'NNP'), ('UNION',
'NNP'), ('January', 'NNP '), (' 31 ', 'CD'), (',', ','), ('
2006 ', 'CD'), ('THE', 'DT'), ('PRESIDENT', 'NNP '), (':',
':'), ('Thank', 'NNP'), ('you', 'PRP'), ('all', 'DT'),
('.', '. ')] [('Mr.', 'NNP'), ('Speaker', 'NNP'), (',',
','), ('Vice', 'NNP'), ('President', 'NNP'), ('Cheney',
'NNP'), (',', ','), ('members', 'NNS'), ('of', 'IN'),
('Congress', 'NNP'), (',', ','), ('members', 'NNS'), ('of',
'IN'), ('the', 'DT'), ('Supreme', 'NNP'), ('Court', 'NNP'),
('and', 'CC'), ('diplomatic', 'JJ'), ('corps', 'NNS'),
(',', ','), ('distinguished', 'VBD'), ('guests', 'NNS'),
(',', ','), ('and', 'CC'), ('fellow', 'JJ'), ('citizens',
'NNS'), (':', ':'), ('Today', 'NN'), ('our', 'PRP  $'),
('nation', 'NN'), ('lost', 'VBD'), ('a', 'DT'), ('beloved',
'VBN'), (',', ' ,'), ('graceful', 'JJ'), (',', ','),
('courageous', 'JJ'), ('woman', 'NN'), ('who', ' WP'),
('called', 'VBN'), ('America', 'NNP'), ('to', 'TO'),
('its', 'PRP$'), ('founding', 'NN'), ('ideals', 'NNS'),
('and', 'CC'), ('carried', 'VBD'), ('on', 'IN'), ('a',
'DT'), ('noble', 'JJ'), ('dream', 'NN'), ('.', '.')]
[('Tonight', 'NNP'), ('we' , 'PRP'), ('are', 'VBP'),
('comforted', 'VBN'), ('by', 'IN'), ('the', 'DT'), ('hope'
, 'NN'), ('of', 'IN'), ('a', 'DT'), ('glad', 'NN'),
('reunion', 'NN'), ('with' , 'IN'), ('the', 'DT'),
('husband', 'NN'), ('who', 'WP'), ('was', 'VBD'), ('taken'
, 'VBN'), ('so', 'RB'), ('long', 'RB'), ('ago', 'RB'),
(',', ','), ('and' , 'CC'), ('we', 'PRP'), ('are', 'VBP'),
('grateful', 'JJ'), ('for', 'IN'), ('the' , 'DT'), ('good',
'NN'), ('life', 'NN'), ('of', 'IN'), ('Coretta', 'NNP'),
('Scott' , 'NNP'), ('King', 'NNP'), ('.', '.')] [('(',
'NN'), ('Applause', 'NNP'), ('. ', '.'), (')', ':')]
[('Pres  Ident', 'NNP'), ('George', 'NNP'), ('W.', 'NNP'),
('Bush', 'NNP'), ('reacts', 'VBZ'), ( 'to', 'TO'),
('applause', 'VB'), ('during', 'IN'), ('his', 'PRP$'),
('State', 'NNP'), ('of', 'IN'), ('the', 'DT'), ('Union',
'NNP'), ('Address', 'NNP'), ('at', 'IN'), ('the', 'DT'),
('Capitol', 'NNP'), (',', ','), ('Tuesday', 'NNP'), (',',
','), ('Jan', 'NNP'), ('.', '.')]
```

When we get here, we can start to get the meaning, but there is still some work to be done. The next topic we will discuss is chunking, where we follow the part of the word, and divide the words into meaningful groups.

V. NLTK block

Now that we know the part of speech, we can pay attention to the so-called block, which divides the word into meaningful blocks. One of the main goals of blocking is to group so-called "noun phrases". These are phrases that contain one or more words of a noun, which may be descriptive words, a verb, or an adverb. The idea is to combine nouns and words related to them.

In order to block, we combine the part-of-speech tag with a regular expression. Mainly from regular expressions, we want to take advantage of these things:

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

If you need help with regular expressions, see the tutorial linked above. The last thing to note is that the word tag is represented by `<` and `>` . We can also place a regular expression in the tag itself to express "all nouns" (`<N.*>`).

```
Import nltk
From nltk.corpus import state_union
From nltk.tokenize import PunktSentenceTokenizer


Train_text = state_union.raw( "2005-GWBush.txt" )
Sample_text = state_union.raw( "2006-GWBush.txt" )


Custom_sent_tokenizer = PunktSentenceTokenizer(train_text)


Tokenized = custom_sent_tokenizer.tokenize(sample_text)


Def process_content () :
    Try :
        For i in tokenized:
            Words = nltk.word_tokenize(i)
            Tagged = nltk.pos_tag(words)
            chunkGram = r"""Chunk: {<RB.?>*<VB.?>*<NNP>+
<NN>?}"""
            chunkParser = nltk.RegexpParser(chunkGram)
            Chunked = chunkParser.parse(tagged)
            Chunked.draw()


    Except Exception as e:
        Print(str(e))

 Process_content()
```

The main line here is:

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

Split this line apart:

`<RB.?>*` : Zero or more adverbs of any tense, followed by:

`<VB.?>*` : Zero or more verbs of any tense, followed by:

`<NNP>+` : One or more reasonable nouns, followed by:

`<NN>?` : Zero or one noun singular.

Try a fun mix to group the various instances until you feel familiar.

Not covered in the video, but there is also a reasonable task to actually access a specific block. This is rarely mentioned, but depending on what you are doing, this can be an important step. Suppose you print out the block and you will see the following output:

```
( S ( Chunk PRESIDENT/NNP GEORGE/NNP W./NNP BUSH/NNP)
'S/POS ( Chunk ADDRESS/NNP BEFORE/NNP A/NNP JOINT/NNP
SESSION/NNP OF/NNP THE/NNP CONGRESS/NNP ON/ NNP THE/NNP
STATE/NNP OF/NNP THE/NNP UNION/NNP January/NNP) 31 /CD , /,
2006 /CD THE/DT ( Chunk PRESIDENT/NNP ) :/ : ( Chunk
Thank/NNP) you/PRP All/DT ./.)
```

Cool, this helps us visualize, but what if we want to access this data through our program? So what happens here is that our "blocking" variable is an NLTK tree. Each "block" and "non-block" is the "subtree" of the tree. We can refer to them by something like `chunked.subtrees` . Then we can iterate through these subtrees like this:

```
For subtree in chunked.subtrees():
        Print (subtree)
```

Next, we may only care about getting these blocks and ignoring the rest. We can use the `filter` parameter in the `chunked.subtrees()` call.

```
For subtree in chunked.subtrees(filter= lambda t: t.label()
== 'Chunk' ):
        Print(subtree)
```

Now we perform filtering to display the subtree with the label "Block". Keep in mind that this is not a "block" in the NLTK block attribute… this is a literal "block" because this is the label we gave it: `chunkGram = r"""Chunk: {<RB.?>*<VB.?>*<NNP>+<NN>?}"""` .

If we write something like `chunkGram = r"""Pythons: {<RB.?>*<VB.?>*<NNP>+<NN>?}"""` , then we can pass `"Pythons."` Tags to filter. The result should be like this:

```
- ( Chunk PRESIDENT / NNP GEORGE / NNP W . / NNP BUSH / NNP
) ( Chunk ADDRESS / NNP BEFORE / NNP A / NNP JOINT / NNP
SESSION / NNP OF / NNP THE / NNP CONGRESS / NNP ON / NNP
THE / NNP STATE / NNP OF / NNP THE / NNP UNION / NNP
January / NNP ) ( Chunk PRESIDENT / NNP ) ( Chunk Thank /
NNP )
```

The complete code is:

```
Import nltk
From nltk.corpus import state_union
From nltk.tokenize import PunktSentenceTokenizer

Train_text = state_union.raw( "2005-GWBush.txt" )
Sample_text = state_union.raw( "2006-GWBush.txt" )

Custom_sent_tokenizer = PunktSentenceTokenizer(train_text)

Tokenized = custom_sent_tokenizer.tokenize(sample_text)

Def process_content () :
    Try :
        For i in tokenized:
            Words = nltk.word_tokenize(i)
            Tagged = nltk.pos_tag(words)
            chunkGram = r"""Chunk: {<RB.?>*<VB.?>*<NNP>+
<NN>?}"""
            chunkParser = nltk.RegexpParser(chunkGram)
            Chunked = chunkParser.parse(tagged)

            Print(chunked)
            For subtree in chunked.subtrees(filter= lambda
t: t.label() == 'Chunk' ):
                Print(subtree)

            Chunked.draw()

    Except Exception as e:
        Print(str(e))

Process_content()
```

VI. NLTK adds a gap (Chinking)

You may find that after a lot of chunking, there are some words in your block that you don't want, but you don't know how to get rid of them by blocking. You may find that adding a gap is your solution.

Adding a gap is similar to a block, which is basically a way to remove a block from a block. The block you removed from the block is your gap.

The code is very similar, you only need to use `}{` to code the gap, behind the block, not the `{}` block.

```
Import nltk
From nltk.corpus import state_union
From nltk.tokenize import PunktSentenceTokenizer

Train_text = state_union.raw( "2005-GWBush.txt" )
Sample_text = state_union.raw( "2006-GWBush.txt" )

Custom_sent_tokenizer = PunktSentenceTokenizer(train_text)

Tokenized = custom_sent_tokenizer.tokenize(sample_text)

Def process_content () :
    Try :
        For i in tokenized[ 5 :]:
            Words = nltk.word_tokenize(i)
            Tagged = nltk.pos_tag(words)

            chunkGram = r"""Chunk: {<.*>+} }<VB.?
|IN|DT|TO>+{"""

            chunkParser = nltk.RegexpParser(chunkGram)
            Chunked = chunkParser.parse(tagged)

            Chunked.draw()

    Except Exception as e:
        Print(str(e))

Process_content()
```

Now, the main difference is:

```
}<VB.?| IN |DT| TO >+{
```

This means that we want to remove one or more verbs, prepositions, qualifiers or `to` words from the gap.

Now that we have learned how to perform some custom partitioning and adding gaps, let's discuss the block form that comes with NLTK, which is named entity recognition.

VII. NLTK named entity recognition

One of the most important forms of blocking in natural language processing is called “named entity recognition.” The idea is to let the machine immediately pull out “entities” such as people, places, things, locations, currencies, and more.

This can be a challenge, but NLTK is built for us. NLTK’s named entity recognition has two main options: identifying all named entities, or identifying named entities as their respective types, such as person, location, location, etc.

This is an example:

```
Import nltk
From nltk.corpus import state_union
From nltk.tokenize import PunktSentenceTokenizer


Train_text = state_union.raw( "2005-GWBush.txt" )
Sample_text = state_union.raw( "2006-GWBush.txt" )

Custom_sent_tokenizer = PunktSentenceTokenizer(train_text)


Tokenized = custom_sent_tokenizer.tokenize(sample_text)

Def process_content () :
    Try :
        For i in tokenized[ 5 :]:
            Words = nltk.word_tokenize(i)
            Tagged = nltk.pos_tag(words)
            namedEnt = nltk.ne_chunk(tagged, binary= True )
            namedEnt.draw()
    Except Exception as e:
        Print(str(e))



Process_content()
```

You can see something right away. When `binary` is false, it also picks the same thing, but breaks the term `White House` into `White` and

`House` as if they are different, and we can see it in the option `binary = True` , named entity The identification says that `White House` is part of the same named entity, which is correct.

According to your goal, you can use the `binary` option. If your `binary` is `false` , here is what you can get, the type of named entity:

```
NE Type and Examples
ORGANIZATION - Georgia -Pacific Corp . , WHO
PERSON - Eddy Bonte, President Obama
LOCATION - Murray River, Mount Everest
DATE - June, 2008 - 06 - 29
TIME - two fifty am, 1 : 30 p . m .
MONEY - 175 million Canadian Dollars, GBP 10.40
PERCENT - twenty pct, 18.75 %
FACILITY - Washington Monument, Stonehenge
GPE - South East Asia, Midlothian
```

Either way, you may find that you need to do more work to get it right, but this feature is very powerful.

In the next tutorial, we will discuss something similar to stem extraction, called “lemmatizing”.

VIII. NLTK morphological restoration

An operation that is very similar to stemming is called a morphological restoration. The main difference between the two is that, as you have seen before, stemming powers often create words that don’t exist, while word forms are actual words.

So, your stem, that is, the word you end up with, is not something you can look up in the dictionary, but you can find a word form.

Sometimes you end up with very similar words, but sometimes you get completely different words. Let’s look at some examples.

```
From nltk.stem import WordNetLemmatizer
```

```
Lemmatizer = WordNetLemmatizer()

  Print(lemmatizer.lemmatize( "cats" ))
  Print(lemmatizer.lemmatize( "cacti" ))
  Print(lemmatizer.lemmatize( "geese" ))
  Print(lemmatizer.lemmatize( "rocks" ))
  Print(lemmatizer.lemmatize( "python" ))
  Print(lemmatizer.lemmatize( "better" , pos= "a" ))
  Print(lemmatizer.lemmatize( "best" , pos= "a" ))
  Print(lemmatizer.lemmatize( "run" ))
  Print(lemmatizer.lemmatize( "run" , 'v' ))
```

Here we have some examples of the form of the word we use. The only thing to note is that `lemmatize` accepts the part-of-speech parameter `pos` . If not provided, the default is "noun". This means that it will try to find the closest noun, which can cause you trouble. If you use morphological restoration, please remember!

In the next tutorial, we'll dive into the NTLK corpus that comes with the module to see all the great documentation, and they are waiting for us there.

IX. NLTK Corpus

In this part of the tutorial, I want to take a moment to dive into the corpus we downloaded all! The NLTK corpus is a collection of natural language data that is definitely worth a look.

Almost all files in the NLTK corpus follow the same rules, accessing them by using the NLTK module, but they are nothing magical. Most of these files are plain text files, some of which are XML files, others are other format files, but can be accessed manually or in modules and Python. Let's talk about viewing them manually.

Depending on your installation, your `nltk_data` directory may be hidden in multiple locations. To find out where it is, go to your Python directory, where the NLTK module is located. If you don't know where, please use the following code:

```
Import nltk
Print (nltk. __file__ )
```

Run it and the output will be the location of the NLTK module `__init__.py` . `data.py` to the NLTK directory and look for the `data.py` file.

The important part of the code is:

```
If sys.platform.startswith( 'win' ):
    # Common locations on Windows:
    Path += [
       Str( r'C:\nltk_data' ), str( r'D:\nltk_data' ), str(
r'E:\nltk_data' ),
        Os.path.join(sys.prefix, str( 'nltk_data' )),
        Os.path.join(sys.prefix, str( 'lib' ), str(
'nltk_data' )),
        Os.path.join(os.environ.get(str( 'APPDATA' ), str(
'C:\\' )), str( 'nltk_data' ))
     ]
Else :
    # Common locations on UNIX & OS X:
    Path += [
        Str( '/usr/share/nltk_data' ),
        Str( '/usr/local/share/nltk_data' ),
        Str( '/usr/lib/nltk_data' ),
        Str( '/usr/local/lib/nltk_data' )
    ]
```

There, you can see the various possible directories for `nltk_data` . If you're on Windows, it's probably in your `appdata` , in a local directory. To do this, you need to open your file browser, go to the top, and type `%appdata%` .

Next click on `roaming` and find the `nltk_data` directory. There, you will find your corpus file. The complete path is like this:

```
C: \Users \swayam.mittal\AppData \Roaming \nltk _data
\corpora
```

Here you have all available corpora, including books, chats, movie reviews and more.

We will now discuss accessing these documents through NLTK. As you can see, these are primarily text documents, so you can open and read documents using plain Python code. That said, the NLTK module has some nice ways to handle the corpus, so you may find that the way to use them is practical. Here's an example of how we opened the Gutenberg Bible and read the first few lines:

```
From nltk.tokenize import sent_tokenize,
PunktSentenceTokenizer
From nltk.corpus import gutenberg

# sample text
Sample = gutenberg.raw( "bible-kjv.txt" )

Tok = sent_tokenize(sample)

For x in range( 5 ):
    Print(tok[x])
```

One of the more advanced data sets is `wordnet` . Wordnet is a collection of words, definitions, examples of their use, synonyms, antonyms, and so on. Next we will use wordnet in depth.

X. NLTK and Wordnet

WordNet is an English vocabulary database created by Princeton and part of the NLTK corpus.

You can use WordNet and NLTK modules together to find word meanings, synonyms, antonyms, and more. Let's introduce some examples.

First, you will need to import `wordnet` :

```
From nltk.corpus import wordnet
```

Then we plan to use the word `program` to find synonyms:

```
Syns = wordnet.synsets( "program" )
```

An example of a synonym:

```
Print (syns[ 0 ].name())

# plan.n.01
```

Just the word:

```
Print(syns[ 0 ] .lemmas ()[ 0 ] .name ())

# plan
```

The definition of the first synonym:

```
Print(syns[ 0 ].definition())

# a series of steps to be carried out or goals to be
accomplished
```

Example of the use of words:

```
Print(syns[ 0 ].examples())


# [ 'they drew up a six-step plan', 'they discussed plans
for a new bond issue']
```

Next, how do we identify synonyms and antonyms of a word? These forms are synonymous, and then you can use `.antonyms` find the antonym of the form. So we can populate some lists like:

```
Synonyms = []
Antonyms = []


For syn in wordnet.synsets( "good" ):
    For l in syn.lemmas():
        Synonyms.append(l.name())
        If l.antonyms():
            Antonyms.append(l.antonyms()[ 0 ].name())


 Print( set (synonyms))
 Print( set (antonyms))
```

```
 '' ' {' beneficial ', ' just ', ' upright ', ' thoroughly
', ' in _force ', ' well ', ' skilful ', ' skillful ', '
sound ', ' unspoiled ', ' expert ', ' Proficient ', ' in
_effect ', ' honorable ', ' adept ', ' secure ', '
commodity ', ' estimable ', ' soundly ', ' right ', '
respectable ', ' good ', ' serious ', ' ripe ', ' salutary
', ' dear ', ' practiced ', ' goodness ', ' safe ', '
effective ', ' unspoilt ', ' dependable ', ' undecomposed
', ' honest ', ' full ', ' near ', ' trade_good '} {' evil
', ' evilness ', ' bad ', ' badness ', ' ill '} ' ''
```

As you can see, our synonym is more than an antonym, because we only look up the antonym of the first form, but you can easily balance this by performing the exact same process for the word `bad` .

Next, we can easily use WordNet to compare the similarities of two words and their tenses, and combine the Wu and Palmer methods for semantic relevance.

Let’s compare nouns `ship` and `boat` :

```
W1 = wordnet.synset( 'ship.n.01' )
W2 = wordnet.synset( 'boat.n.01' )
print (w1.wup_similarity(w2))

# 0.9090909090909091

W1 = wordnet.synset( 'ship.n.01' )
W2 = wordnet.synset( 'car.n.01' )
print (w1.wup_similarity(w2))

# 0.6956521739130435

W1 = wordnet.synset( 'ship.n.01' )
W2 = wordnet.synset( 'cat.n.01' )
print (w1.wup_similarity(w2))

# 0.38095238095238093
```

Next, we will discuss some issues and start discussing the topic of text categorization.

XI. NLTK text classification

Now that we are familiar with NLTK, let’s try to deal with text categorization. The goal of text categorization can be quite broad. Maybe we are trying to classify the text as political or military. Maybe we try to classify by the gender of the author. A fairly popular text categorization task is to identify the body of the text as spam or non-spam, such as an email filter. In our case, we will try to create a sentiment analysis algorithm.

To do this, we first try to use a movie review database that belongs to the NLTK corpus. From there, we will try to use vocabulary as a “feature”, which is part of a “positive” or “negative” movie review. The NLTK corpus `movie_reviews` dataset has comments and they are marked as positive or negative. This means we can train and test this data. First, let's preprocess our data.

```
Import nltk
import random
```

```
from nltk.corpus import movie_reviews


Documents = [(list(movie_reviews.words(fileid)), category)
             For category in movie_reviews.categories()
              for fileid in
movie_reviews.fileids(category)]


Random.shuffle(documents)


Print(documents[ 1 ])


All_words = []
For w in movie_reviews.words():
    All_words.append(w.lower())


All_words = nltk.FreqDist(all_words)
Print(all_words.most_common( 15 ))
Print(all_words[ "stupid" ])
```

It may take some time to run this script because the movie review dataset is a bit large. Let us introduce what happened here.

After importing the data set we want, you will see:

```
Documents = [( list (movie_reviews. words (fileid)),
category)
              for category in movie_reviews.categories()
              for fileid in
movie_reviews.fileids(category)]
```

Basically, in simple English, the above code is translated into: in each category (we have forward and exclusive), select all file IDs (each comment has its own ID), then store the `word_tokenized` version for the file ID (word list) followed by a positive or negative label in a large list.

Next, we use `random` to disrupt our files. This is because we are going to train and test. If we order them in order, we may train all negative comments, and some positive comments, and then test on all positive comments. We don't want this, so we messed up the data.

Then, in order for you to see the data you are using, we print out `documents[1]` that this is a large list where the first element is a list of words and the second element is `pos` or `neg` label.

Next, we want to collect all the words we found, so we can have a huge list of typical words. From here, we can perform a frequency distribution and then find the most common words. As you can see, the most popular “words” are actually punctuation, `the` , `a` etc., but soon we will get valid vocabulary. We are going to store thousands of the most popular words, so this shouldn't be a problem.

```
Print(all_words. most_common( 15 ) )
```

The 15 most commonly used words are given above. You can also find out the number of occurrences of a word by following the steps below:

```
Print  (all_words[ "stupid" ])
```

Next, we begin to store our words as features of positive or negative movie reviews.

XII. use NLTK to convert words into features

In this tutorial, we build on previous videos and compile a list of features for words in positive and negative comments to see trends in specific types of words in positive or negative comments.

Initially, our code:

```
Import nltk
Import random
from nltk.corpus import movie_reviews
```

```
Documents = [(list(movie_reviews. words (fileid)),
category)
              for category in movie_reviews.categories()
              for fileid in
movie_reviews.fileids(category)]


Random .shuffle(documents)


All_words = []


For w in movie_reviews. words ():
    All_words.append(w. lower ())


All_words = nltk.FreqDist(all_words)


Word_features = list(all_words. keys ())[: 3000 ]
```

Almost as before, just now there is a new variable that `word_features` contains the top 3000 most commonly used words. Next, we'll build a simple function that finds these top 3000 words in our positive and negative documents, marking their presence as yes or no:

```
Def  find_features  (document) :
    Words = set(document)
    Features = {}
    For w in word_features:
        Features[w] = (w in words)


    Return features
```

Below, we can print out the feature set:

```
Print(( find_features(movie_reviews. words(
'neg/cv000_29416.txt' ) ) ) ) )
```

We can then do this for all of our documentation by saving the feature boolean values and their respective positive or negative categories by doing the following:

```
Featuresets = [(find_features(rev), category) for (rev,
category) in documents]
```

Awesome, now we have features and tags, what's next? Usually, the next step is to continue and train the algorithm and then test it. So let's continue to do this, starting with the naive Bayes classifier in the next tutorial!

XIII. NLTK Naive Bayes Classifier

It's time to choose an algorithm that divides our data into training and test sets and then starts! The algorithm we first use is the naive Bayes classifier. This is a very popular text categorization algorithm, so we can only try it first. However, before we can train and test our algorithm, we need to first break the data into a training set and a test set.

You can train and test the same data set, but this will give you some serious deviations, so you shouldn't train and test the exact same data. For this reason, since we have disrupted the data set, we will first use 1900 out-of-order comments with positive and negative comments as the training set. Then we can test on the last 100 to see how accurate we are.

This is called supervised machine learning because we are presenting data to the machine and telling it "this data is positive" or "this data is negative." Then, after the training is complete, we show the machine some new data and ask the computer what we think of the new data based on what we have taught before.

We can split the data in the following ways:

```
# set that we'll train our classifier with
training_ set = featuresets[: 1900 ]

# set that we'll test against.
testing_ set = featuresets[ 1900 :]
```

Below, we can define and train our classifiers:

```
Classifier = nltk .NaiveBayesClassifier  .train
(training_set)
```

First, we simply call the naive Bayes classifier and use it in a row `.train()` for training.

It's simple enough, now it's trained. Next, we can test it:

```
Print ( "Classifier accuracy percent:" ,
(nltk.classify.accuracy(classifier, testing_set)) *100 )
```

Hey, you got your answer. If you missed it, the reason we can "test" the data is that we still have the right answer. So, in the test, we present the data to the computer without providing the correct answer. If it correctly guesses what we know, then the computer is correct. Considering the disruption we've done, you and I may be different in accuracy, but you should see an average of 60–75% accuracy.

Next, we can learn more about the most valuable words in positive or negative comments:

```
classifier the .Show _most_informative_features ( 15 )
```

This is different for everyone, but you should see something like this:

```
Most Informative Features
Insulting = True neg : pos = 10.6 : 1.0
ludicrous = True neg : pos = 10.1 : 1.0
winslet = True pos : neg = 9.0 : 1.0
detract = True pos : neg = 8.4 : 1.0
breathtaking = True pos : neg = 8.1 : 1.0
```

```
Silverstone = True neg : pos = 7.6 : 1.0
excruciatingly = True neg : pos = 7.6 : 1.0
warns =True pos : neg = 7.0 : 1.0
tracy = True pos : neg = 7.0 : 1.0
insipid = True neg : pos = 7.0 : 1.0
freddie = True neg : pos = 7.0 : 1.0
damon = True pos : neg = 5.9 : 1.0
debate = True pos : neg = 5.9 : 1.0
ordered = True pos : neg = 5.8 : 1.0
lang = True pos : neg = 5.7: 1.0
```

What this tells you is that each word's negative to positive appearance, or vice versa. So here we can see that the `insulting` word in the negative comment appears 10.6 times more than the positive comment. `Ludicrous` It is 10.1.

Now let's assume that you are completely satisfied with your results, that you want to continue, and perhaps use this classifier to predict what is going on. It is very impractical to train the classifier and retrain it whenever you need to use the classifier. Therefore, you can use the `pickle` module to save the classifier. We will do it next.

XIV. Save the classifier with NLTK

Training classifiers and machine learning algorithms can take a long time, especially if you train on a larger data set. Ours is actually very small. Can you imagine that you have to train the classifier every time you want to start using the classifier? So horrible! Instead, we can use the `pickle` module and serialize our classifier object so that all we have to do is simply load the file.

So what should we do? The first step is to save the object. To do this, you first need to import at the top of the script `pickle` , and then after `.train()` training with the classifier, you can call the following lines:

```
Save_classifier = open ( "naivebayes.pickle" , "wb" )
Pickle. dump (classifier, save_classifier)
Save_classifier. close ()
```

This opens a `pickle` file and is ready to write some data in bytes. Then we use `pickle.dump()` to dump the data. `pickle.dump()` The first argument is what you write, and the second argument is where you write it.

After that, we closed the file as we requested, which means that we now have a `pickle` serialized object in the script's directory!

Next, how do we start using this classifier? `.pickle` Files are serialized objects, all we need to do now is to read them into memory, which is as simple as reading any other normal file. this way:

```
Classifier_f = open ( "naivebayes.pickle" , "rb" )
Classifier = pickle. load (classifier_f)
Classifier_f. close ()
```

Here we have performed a very similar process. We open the file to read the bytes. Then we use `pickle.load()` to load the file and save the data to the classifier variable. Then we close the file, that's it. We now have the same classifier object as before!

Now we can use this object, and whenever we want to use it for classification, we no longer need to train our classifier.

Although it’s all good, we may not be happy with the 60–75% accuracy we get. What about other classifiers? In fact, there are many classifiers, but we need the scikit-learn (sklearn) module. Fortunately, NLTK employees recognized the value of incorporating the sklearn module into NLTK, and they built a small API for us. This is what we will do in the next tutorial.

XV. NLTK and Sklearn

Now we have seen how easy it is to use a classifier, now we want to try more! The best module for Python is the Scikit-learn (sklearn) module.

If you want to learn more about the Scikit-learn module, I have some tutorials on Scikit-Learn machine learning.

Fortunately, for us, the people behind NLTK value the value of incorporating the sklearn module into the NLTK classifier approach. In this way, they created various `SklearnClassifier` APIs. To use it, you just need to import it like this:

```
From nltk.classify.scikitlearn import SklearnClassifier
```

From here on, you can use any `sklearn` classifier. For example, let's introduce more variants of the naive Bayesian algorithm:

```
From sklearn.naive_bayes import MultinomialNB , BernoulliNB
```

After how do you use them? As a result, this is very simple.

```
MNB_classifier = SklearnClassifier(MultinomialNB())
MNB_classifier .train (training_set)
Print( " MultinomialNB accuracy percent:" ,nltk .classify
.accuracy (MNB_classifier, testing_set))

BNB_classifier = SklearnClassifier(BernoulliNB())
BNB_classifier .train (training_set)
Print( "BernoulliNB accuracy percent:" ,nltk .classify
.accuracy (BNB_classifier, testing_set))
```

It’s that simple. Let us introduce more things:

```
From sklearn.linear_model import
LogisticRegression,SGDClassifier
 from sklearn.svm import SVC, LinearSVC, NuSVC
```

Now all our classifiers should look like this:

```
Print( "Original Naive Bayes Algo accuracy percent:" ,
(nltk .classify  .accuracy (classifier, testing_set))* 100
)
classifier the .Show _most_informative_features ( 15 )
```

```
MNB_classifier = SklearnClassifier(MultinomialNB())
MNB_classifier .train (training_set)
Print( "MNB_classifier accuracy percent:" , (nltk .classify
.accuracy (MNB_classifier, testing_set))* 100 )


BernoulliNB_classifier = SklearnClassifier(BernoulliNB())
BernoulliNB_classifier .train (training_set)
Print( "BernoulliNB_classifier accuracy percent:" , (nltk
.classify  .accuracy (BernoulliNB_classifier,
testing_set))* 100 )


LogisticRegression_classifier =
SklearnClassifier(LogisticRegression())
LogisticRegression_classifier .train (training_set)
Print( "LogisticRegression_classifier accuracy percent:" ,
(nltk .classify  .accuracy (LogisticRegression_classifier,
testing_set))* 100 )


SGDClassifier_classifier =
SklearnClassifier(SGDClassifier())
SGDClassifier_classifier .train (training_set)
Print( "SGDClassifier_classifier accuracy percent:" , (nltk
.classify  .accuracy (SGDClassifier_classifier,
testing_set))* 100 )


SVC_classifier = SklearnClassifier(SVC())
SVC_classifier .train (training_set)
Print( "SVC_classifier accuracy percent:" , (nltk .classify
.accuracy (SVC_classifier, testing_set))* 100 )


LinearSVC_classifier = SklearnClassifier(LinearSVC())
LinearSVC_classifier .train (training_set)
Print( "LinearSVC_classifier accuracy percent:" , (nltk
.classify  .accuracy (LinearSVC_classifier, testing_set))*
100 )


NuSVC_classifier = SklearnClassifier(NuSVC())
NuSVC_classifier .train (training_set)
Print( "NuSVC_classifier accuracy percent:" , (nltk
.classify  .accuracy (NuSVC_classifier, testing_set))* 100
)
```

The result of running it should look like this:

```
Original Naive Bayes Algo accuracy percent: 63.0
Most Informative Features
Thematic = True pos : neg = 9.1 : 1.0
secondly = True pos : neg = 8.5 : 1.0
narrates = True pos : neg = 7.8 : 1.0
rounded = True pos : neg = 7.1 : 1.0
supreme = True pos : neg = 7.1 : 1.0
Layered = True pos : neg = 7.1 : 1.0
crappy = True Neg : pos = 6.9 : 1.0
uplifting = True pos : neg = 6.2 : 1.0
ugh = True neg : pos = 5.3 : 1.0
mamet = True pos : neg = 5.1 : 1.0
gaining = True pos : neg = 5.1 : 1.0
wanda = True Neg : pos = 4.9 : 1.0
onset = True neg : pos = 4.9 :1.0
fantastic = True pos : neg = 4.5 : 1.0
kentucky = True pos : neg = 4.4 : 1.0
MNB_classifier accuracy percent: 66.0
BernoulliNB_classifier accuracy percent: 72.0
LogisticRegression_classifier accuracy percent: 64.0
SGDClassifier_classifier accuracy percent: 61.0
SVC_classifier accuracy percent: 45.0
LinearSVC_classifier accuracy percent: 68.0
NuSVC_classifier accuracy percent: 59.0
```

So, we can see that SVC errors are more common than correct, so we should probably discard it. but? Next we can try to use all of these algorithms at once. An algorithmic algorithm! To do this, we can create another classifier and generate the results of the classifier based on the results of other algorithms. It's a bit like a voting system, so we only need an odd number of algorithms. This is what we will discuss in the next tutorial.

XVI. Using the NLTK combination algorithm

Now we know how to use a bunch of algorithm classifiers, like a child on Candy Island, telling them that they can only choose one, and we may find it difficult to select only one classifier. The good news is that you don't have to! The combined classifier algorithm is a commonly used technique, which is implemented by creating a voting system. Each algorithm has one vote and the most votes are selected.

To do this, we want our new classifier to work like a typical NLTK classifier and have all the methods. Quite simply, with object-oriented programming, we can ensure that we inherit from the NLTK classifier class. To do this we will import it:

```
From nltk.classify import ClassifierI
from statistics import mode
```

We also import `mode` (the majority) because this will be the way we choose the maximum count.

Now let's build our classifier class:

```
Class  VoteClassifier  (ClassifierI) :
    def  __init__  (self, *classifiers) :
        self._classifiers = classifiers
```

We call our class `VoteClassifier` , we inherit NLTK `ClassifierI` . Next, we assign the list of classifiers passed to our class `self._classifiers` .

Next, we will continue to create our own classification method. We intend to call it `.classify` so that we can call it later `.classify` , just like the traditional NLTK classifier.

```
Def  classify  (self, features) :
        Votes = []
        For c in self._classifiers:
            v = c.classify(features)
            Votes.append(v)
        Return mode(votes)
```

Quite simply, what we are doing here is to iterate through our list of classifier objects. Then, for each one, we ask it to be based on feature classification. Classification is considered a vote. After the traversal is complete, we return `mode(votes)` , this is just the mode to return the vote.

This is what we really need, but I think another parameter, confidence is useful. Since we have a voting algorithm, we can also count the number of support and negative votes, and call it "confidence." For example, the confidence of a 3/5 vote is weaker than a 5/5 vote. Therefore, we can literally return the voting ratio as a measure of confidence. This is our confidence method:

```
Def  confidence  (self, features) :
        Votes = []
        For c in self._classifiers:
            v = c.classify(features)
            Votes.append(v)

        Choice_votes = votes.count(mode(votes))
        Conf = choice_votes / len(votes)
        Return conf
```

Now let's put things together:

```
Import nltk
Import random
From nltk.corpus import movie_reviews
From nltk.classify.scikitlearn import SklearnClassifier
Import pickle

From sklearn.naive_bayes import MultinomialNB, BernoulliNB
From sklearn.linear_model import LogisticRegression,
SGDClassifier
From sklearn.svm import SVC, LinearSVC, NuSVC


From nltk.classify import ClassifierI
From statistics import mode



Class VoteClassifier(ClassifierI):
Def __init__(self, *classifiers):
self._classifiers = classifiers

Def classify(self, features):
votes = []
for c in self._classifiers:
```

```
v = c.classify(features)
votes.append(v)
return mode(votes)


Def confidence(self, features):
votes = []
for c in self._classifiers:
v = c.classify(features)
votes.append(v)


Choice_votes = votes.count(mode(votes))
conf = choice_votes / len(votes)
return conf
```

```
Documents = [(list(movie_reviews.words(fileid)), category)
For category in movie_reviews.categories()
for fileid in movie_reviews.fileids(category)]


Random.shuffle(documents)


All_words = []
```

```
For w in movie_reviews.words():
 All_words.append(w.lower())


All _words = nltk.FreqDist(all_ words)


Word _features = list(all_ words.keys())[:3000]
```

```
Def find_features(document):
Words = set(document)
features = {}
for w in word_features:
features[w] = (w in words)


 Return features


#print((find_features(movie_reviews.words('neg/cv000_29416.
txt'))))
```

```
Featuresets = [(find_features(rev), category) for (rev,
category) in documents]


Training_set = featuresets[:1900]
Testing_set = featuresets[1900:]


#classifier = nltk.NaiveBayesClassifier.train(training_set)
```

```
Classifier_f = open("naivebayes.pickle","rb")
Classifier = pickle.load(classifier_f)
Classifier_f.close()
```

```
Print("Original Naive Bayes Algo accuracy percent:",
(nltk.classify.accuracy(classifier, testing_set))*100)
classifier.show _most_ informative_features (15)


MNB_classifier = SklearnClassifier(MultinomialNB())
MNB _classifier.train(training_ set)
Print("MNB _classifier accuracy percent:",
(nltk.classify.accuracy(MNB_ classifier, testing_set))*100)


BernoulliNB_classifier = SklearnClassifier(BernoulliNB())
BernoulliNB _classifier.train(training_ set)
Print("BernoulliNB _classifier accuracy percent:",
(nltk.classify.accuracy(BernoulliNB_ classifier,
testing_set))*100)


LogisticRegression_classifier =
SklearnClassifier(LogisticRegression())
LogisticRegression _classifier.train(training_ set)
Print("LogisticRegression _classifier accuracy percent:",
(nltk.classify.accuracy(LogisticRegression_ classifier,
testing_set))*100)


SGDClassifier_classifier =
SklearnClassifier(SGDClassifier())
SGDClassifier _classifier.train(training_ set)
Print("SGDClassifier _classifier accuracy percent:",
(nltk.classify.accuracy(SGDClassifier_ classifier,
testing_set))*100)

##SVC_classifier = SklearnClassifier(SVC())
##SVC_classifier.train(training_set)
##print("SVC_classifier accuracy percent:",
(nltk.classify.accuracy(SVC_classifier, testing_set))*100)


LinearSVC_classifier = SklearnClassifier(LinearSVC())
LinearSVC _classifier.train(training_ set)
Print("LinearSVC _classifier accuracy percent:",
(nltk.classify.accuracy(LinearSVC_ classifier,
testing_set))*100)


NuSVC_classifier = SklearnClassifier(NuSVC())
NuSVC _classifier.train(training_ set)
Print("NuSVC _classifier accuracy percent:",
(nltk.classify.accuracy(NuSVC_ classifier,
testing_set))*100)



Voted_classifier = VoteClassifier(classifier,
NuSVC_classifier,
LinearSVC_classifier,
SGDClassifier_classifier,
MNB_classifier,
BernoulliNB_classifier,
LogisticRegression_classifier)
```

```
Print("voted _classifier accuracy percent:",
(nltk.classify.accuracy(voted_ classifier,
testing_set))*100)


Print("Classification:", voted
_classifier.classify(testing_ set[ 0 ][ 0 ]), "Confidence
%:",voted _classifier.confidence(testing_ set[ 0 ][ 0
])*100)
Print("Classification:", voted
_classifier.classify(testing_ set[ 1 ][ 0 ]), "Confidence
%:",voted _classifier.confidence(testing_ set[ 1 ][ 0
])*100)
Print("Classification:", voted
_classifier.classify(testing_ set[ 2 ][ 0 ]), "Confidence
%:",voted _classifier.confidence(testing_ set[ 2 ][ 0
])*100)
Print("Classification:", voted
_classifier.classify(testing_ set[ 3 ][ 0 ]), "Confidence
%:",voted _classifier.confidence(testing_ set[ 3 ][ 0
])*100)
Print("Classification:", voted
_classifier.classify(testing_ set[ 4 ][ 0 ]), "Confidence
%:",voted _classifier.confidence(testing_ set[ 4 ][ 0
])*100)
Print("Classification:", voted
_classifier.classify(testing_ set[ 5 ][ 0 ]), "Confidence
%:",voted _classifier.confidence(testing_ set[ 5 ][ 0
])*100)
```

So at the end, we run some sorter examples for the text. All our output:

```
Original Naive Bayes Algo accuracy percent: 66.0
Most Informative Features
                Thematic = True pos : neg = 9.1 : 1.0
                secondly = True pos : neg = 8.5 : 1.0
                narrates = True pos : neg = 7.8 : 1.0
                 layered = True pos : neg = 7.1 : 1.0
                 rounded = True pos : neg = 7.1 : 1.0
                 Supreme = True pos : neg = 7.1 : 1.0
                  crappy = True neg: pos = 6.9 : 1.0
               uplifting = True pos : neg = 6.2 : 1.0
                     ugh = True neg : pos = 5.3 : 1.0
                 gaining = True pos : neg = 5.1 : 1.0
                   mamet = True pos : neg = 5.1 : 1.0
                   wanda = True neg : pos = 4.9 : 1.0
                   onset = True neg : pos = 4.9 :1.0
               fantastic = True pos : neg = 4.5 : 1.0
                   milos = True pos : neg = 4.4 : 1.0
MNB_classifier accuracy percent: 67.0
BernoulliNB_classifier accuracy percent: 67.0
LogisticRegression_classifier accuracy percent: 68.0
SGDClassifier_classifier accuracy percent:
57.99999999999999
```

```
LinearSVC_classifier accuracy percent: 67.0
NuSVC_classifier accuracy percent: 65.0
voted_classifier accuracy percent: 65.0
Classification:  neg Confidence %:100.0
Classification: pos Confidence %: 57.14285714285714
Classification:  neg Confidence %: 57.14285714285714
Classification:  neg Confidence %: 57.14285714285714
Classification: pos Confidence %: 57.14285714285714
Classification: pos Confidence %: 85.71428571428571
```

XVII. Investigate bias using NLTK

In this tutorial we will discuss some issues. The main problem is that we have a fairly biased algorithm. You can test it by commenting out the scramble of the document, then training with the first 1900 and leaving the last 100 (all positive) comments. Test it and you will find that your accuracy is very poor.

Instead, you can test with the top 100 data, all of which are negative and use 1900 training. Here you will find that the accuracy is very high. This is a bad sign. This can mean a lot of things, and we have a lot of options to solve it.

In other words, the project we are considering suggests that we continue and use different data sets, so we will do so. Finally, we will find that there is still some deviation in this new data set, that is, it chooses negative things more often. The reason is that the negatives of negative reviews tend to be more positive than positive ones. This can be done with some simple weighting, but it can also be complicated. Maybe it's another day's tutorial. Now we are going to grab a new data set, which we will discuss in the next tutorial.

XVIII. Using NLTK to improve training data for sentiment analysis

So now is the time to train on the new data set. Our goal is to analyze Twitter's sentiment, so we want every positive and negative statement in the dataset to be short. It happens that I have 5300+ positive and 5300+ negative movie reviews, which is a much shorter data set. We should be able to get more accuracy from a larger training set and fit Twitter tweets better.

I hosted these two files here, you can find them by downloading a short comment . Save these files as `positive.txt` and `negative.txt` .

Now we can build new data sets as before. What needs to be changed?

We need a new way to create our "document" variable, and then we need a new way to create the `all_words` variable. Really no problem, I did this:

```
Short_pos = open ( "short_reviews/positive.txt" , "r" ).
read ()
Short_neg = open ( "short_reviews/negative.txt" , "r" ).
read ()


Documents = []


For r in short_pos. split ( '\n' ):
    Documents.append( (r, "pos" ) )


For r in short_neg. split ( '\n' ):
    Documents.append( (r, "neg" ) )



All_words = []


Short_pos_words = word_tokenize(short_pos)
Short_neg_words = word_tokenize(short_neg)


For w in short_pos_words:
    All_words.append(w. lower ())


For w in short_neg_words:
    All_words.append(w. lower ())


All_words = nltk.FreqDist(all_words)
```

Next, we also need to adjust our feature lookup function, mainly based on the words in the document, because our new sample has no beautiful `.words()` features. I continued and added the most common words:

```
Word_features = list(all_words.keys())[: 5000 ]


Def  find_features  (document) :
    Words = word_tokenize(document)
    Features = {}
    For w in word_features:
        Features[w] = (w in words)


    Return features


Featuresets = [(find_features(rev), category) for (rev,
category) in documents]
Random.shuffle(featuresets)
```

Other than that, the rest are the same. This is a complete script just in case you or I missed something:

This process takes a while. You may want to do something else. It took me about 30–40 minutes to get it all running, and I ran it on the i7 3930k. At the time of writing this article (2015), a general processor might take several hours. But this is a one-off process.

```
Import nltk
import random
from nltk.corpus import movie_reviews
from nltk.classify.scikitlearn import SklearnClassifier
import pickle


From sklearn.naive_bayes import MultinomialNB , BernoulliNB
from sklearn.linear_model import LogisticRegression,
SGDClassifier
from sklearn.svm import SVC, LinearSVC, NuSVC


From nltk.classify import ClassifierI
from statistics import mode


From nltk.tokenize import word_tokenize



Class  VoteClassifier  (ClassifierI) :
    def  __init__  (self, *classifiers) :
        Self._classifiers = classifiers


    Def  classify  (self, features) :
        Votes = []
        For c in self._classifiers:
            v = c.classify(features)
            Votes.append(v)
        Return mode(votes)


    Def  confidence  (self, features) :
        Votes = []
        For c in self._classifiers:
            v = c.classify(features)
            Votes.append(v)


        Choice_votes = votes.count(mode(votes))
        Conf = choice_votes / len(votes)
        Return conf


Short_pos = open( "short_reviews/positive.txt" , "r"
).read()
Short_neg = open( "short_reviews/negative.txt" , "r"
).read()

Documents = []


For r in short_pos.split( '\n' ):
    Documents.append( (r, "pos" ) )


For r in short_neg.split( '\n' ):
    Documents.append( (r, "neg" ) )



All_words = []


Short_pos_words = word_tokenize(short_pos)
Short_neg_words = word_tokenize(short_neg)


For w in short_pos_words:
    All_words.append(w.lower())

For w in short_neg_words:
    All_words.append(w.lower())


All_words = nltk.FreqDist(all_words)

Word_features = list(all_words.keys())[: 5000 ]


Def  find_features  (document) :
    Words = word_tokenize(document)
    Features = {}
    For w in word_features:
        Features[w] = (w in words)


    Return features
```

```
#print((find_features(movie_reviews.words('neg/cv000_29416.
txt')))))


Featuresets = [(find_features(rev), category) for (rev,
category) in documents]

Random.shuffle(featuresets)


# positive data example:
training_set = featuresets[: 10000 ]
Testing_set = featuresets[ 10000 :]


##
### negative data example:
##training_set = featuresets[100:]
##testing_set = featuresets[:100]



Classifier = nltk.NaiveBayesClassifier.train(training_set)
Print( "Original Naive Bayes Algo accuracy percent:" ,
(nltk.classify.accuracy(classifier, testing_set))* 100 )
Classifier.show_most_informative_features( 15 )


MNB_classifier = SklearnClassifier(MultinomialNB())
MNB_classifier.train(training_set)
Print( "MNB_classifier accuracy percent:" ,
(nltk.classify.accuracy(MNB_classifier, testing_set))* 100
)


BernoulliNB_classifier = SklearnClassifier(BernoulliNB())
BernoulliNB_classifier.train(training_set)
Print( "BernoulliNB_classifier accuracy percent:" ,
(nltk.classify.accuracy(BernoulliNB_classifier,
testing_set))* 100 )

LogisticRegression_classifier =
SklearnClassifier(LogisticRegression())
LogisticRegression_classifier.train(training_set)
Print( "LogisticRegression_classifier accuracy percent:" ,
(nltk.classify.accuracy(LogisticRegression_classifier,
testing_set))* 100 )


SGDClassifier_classifier =
SklearnClassifier(SGDClassifier())
SGDClassifier_classifier.train(training_set)
Print( "SGDClassifier_classifier accuracy percent:" ,
(nltk.classify.accuracy(SGDClassifier_classifier,
testing_set))* 100 )


##SVC_classifier = SklearnClassifier(SVC())
##SVC_classifier.train(training_set)
##print("SVC_classifier accuracy percent:",
(nltk.classify.accuracy(SVC_classifier, testing_set))*100)
```

```
LinearSVC_classifier = SklearnClassifier(LinearSVC())
LinearSVC_classifier.train(training_set)
Print( "LinearSVC_classifier accuracy percent:" ,
(nltk.classify.accuracy(LinearSVC_classifier,
testing_set))* 100 )


NuSVC_classifier = SklearnClassifier(NuSVC())
NuSVC_classifier.train(training_set)
Print( "NuSVC_classifier accuracy percent:" ,
(nltk.classify.accuracy(NuSVC_classifier, testing_set))*
100 )



Voted_classifier = VoteClassifier(
                                  NuSVC_classifier,
                                  LinearSVC_classifier,
                                  MNB_classifier,
                                  BernoulliNB_classifier,

LogisticRegression_classifier)

Print( "voted_classifier accuracy percent:" ,
(nltk.classify.accuracy(voted_classifier, testing_set))*
100 )
```

Output:

```
Original Naive Bayes Algo accuracy percent:
66.26506024096386
Most Informative Features
              Refreshing = True pos : neg = 13.6 : 1.0
                captures = True pos : neg = 11.3 : 1.0
                  stupid = True neg : pos = 10.7 : 1.0
                  tender = True pos : neg = 9.6 : 1.0
              meandering = True neg : pos = 9.1 : 1.0
                      Tv = True neg : pos = 8.6 : 1.0
                 low-key = TruePos : neg = 8.3 : 1.0
              thoughtful = True pos : neg = 8.1 : 1.0
                   banal = True neg : pos = 7.7 : 1.0
              amateurish = True neg : pos = 7.7 : 1.0
                terrific = True pos : neg = 7.6 : 1.0
                  record = True Pos : neg = 7.6 : 1.0
             captivating = True pos : neg = 7.6 :1.0
                portrait = True pos : neg = 7.4 : 1.0
                 culture = True pos : neg = 7.3 : 1.0
MNB_classifier accuracy percent: 65.8132530120482
BernoulliNB_classifier accuracy percent: 66.71686746987952
LogisticRegression_classifier accuracy percent:
67.16867469879519
SGDClassifier_classifier accuracy percent: 65.8132530120482
LinearSVC_classifier accuracy percent: 66.71686746987952
```

```
NuSVC_classifier accuracy percent: 60.09036144578314
voted_classifier accuracy percent: 65.66265060240963
```

Yes, I bet you spent a while, so in the next tutorial we will talk about `pickle` everything!

XIX. Use NLTK to create modules for sentiment analysis

With this new data set and new classifiers, we can move on. As you may have noticed, this new data set takes longer to train because it is a larger collection. I have shown you that by `pickel` serializing or serializing the trained classifiers, we can actually save a lot of time. These classifiers are just objects.

I have shown you how to use it `pickel` to implement it, so I encourage you to try it yourself. If you need help, I will paste the complete code… but be careful, do it yourself!

This process takes a while. You may want to do something else. It took me about 30–40 minutes to get it all running, and I ran it on the i7 3930k. At the time of writing this article (2015), a general processor might take several hours. But this is a one-off process.

```
Import nltk
 import random
 #from nltk.corpus import movie_reviews
from nltk.classify.scikitlearn import SklearnClassifier
 import pickle
 from sklearn.naive_bayes import MultinomialNB ,
BernoulliNB
 from sklearn.linear_model import LogisticRegression,
SGDClassifier
 from sklearn.svm import SVC, LinearSVC, NuSVC
 from nltk. Classify import ClassifierI
 from statistics import mode
 from nltk.tokenize import Word_tokenize


Class  VoteClassifier  (ClassifierI) :
    def  __init__  (self, *classifiers) :
        Self._classifiers = classifiers
    Def  classify  (self, features) :
        Votes = []
        For c in self._classifiers:
            v = c.classify(features)
            Votes.append(v)
        Return mode(votes)


    Def  confidence  (self, features) :
        Votes = []
        For c in self._classifiers:
            v = c.classify(features)
            Votes.append(v)


        Choice_votes = votes.count(mode(votes))
        Conf = choice_votes / len(votes)
        Return conf


Short_pos = open( "short_reviews/positive.txt" , "r"
).read()
Short_neg = open( "short_reviews/negative.txt" , "r"
).read()


# move this up here
All_words = []
Documents = []



# j is adject, r is adverb, and v is verb
#allowed_word_types = ["J","R","V"]
allowed_word_types = [ "J" ]


For p in short_pos.split( '\n' ):
    Documents.append( (p, "pos" ) )
    Words = word_tokenize(p)
    Pos = nltk.pos_tag(words)
    For w in pos:
         if w[ 1 ][ 0 ] in allowed_word_types:
            All_words.append(w[ 0 ].lower())



For p in short_neg.split( '\n' ):
    Documents.append( (p, "neg" ) )
    Words = word_tokenize(p)
    Pos = nltk.pos_tag(words)
    For w in pos:
         if w[ 1 ][ 0 ] in allowed_word_types:
            All_words.append(w[ 0 ].lower())



Save_documents
= open( "pickled_algos/documents.pickle" , "wb" )
Pickle.dump(documents, save_documents)
Save_documents.close()
```

```
All_words = nltk.FreqDist(all_words)



Word_features = list(all_words.keys())[: 5000 ]



Save_word_features = open(
"pickled_algos/word_features5k.pickle" , "wb" )
Pickle.dump(word_features, save_word_features)
Save_word_features.close()



Def  find_features  (document) :
    Words = word_tokenize(document)
    Features = {}
    For w in word_features:
        Features[w] = (w in words)

    Return features


Featuresets = [(find_features(rev), category) for (rev,
category) in documents]


Random.shuffle(featuresets)
Print(len(featuresets))

Testing_set = featuresets[ 10000 :]
Training_set = featuresets[: 10000 ]



Classifier = nltk.NaiveBayesClassifier.train(training_set)
Print( "Original Naive Bayes Algo accuracy percent:" ,
(nltk.classify.accuracy(classifier, testing_set))* 100 )
Classifier.show_most_informative_features( 15 )


###############
save_classifier = open(
"pickled_algos/originalnaivebayes5k.pickle" , "wb" )
Pickle.dump(classifier, save_classifier)
Save_classifier.close()


MNB_classifier = SklearnClassifier(MultinomialNB())
MNB_classifier.train(training_set)
Print( "MNB_classifier accuracy percent:" ,
(nltk.classify.accuracy(MNB_classifier, testing_set))* 100
)


Save_classifier = open(
"pickled_algos/MNB_classifier5k.pickle" , "wb" )
Pickle.dump(MNB_classifier, save_classifier)
Save_classifier.close()
```

```
BernoulliNB_classifier = SklearnClassifier(BernoulliNB())
BernoulliNB_classifier.train(training_set)
Print( "BernoulliNB_classifier accuracy percent:" ,
(nltk.classify.accuracy(BernoulliNB_classifier,
testing_set))* 100 )


Save_classifier = open(
"pickled_algos/BernoulliNB_classifier5k.pickle" , "wb" )
Pickle.dump(BernoulliNB_classifier, save_classifier)
Save_classifier.close()


LogisticRegression_classifier =
SklearnClassifier(LogisticRegression())
LogisticRegression_classifier.train(training_set)
Print( "LogisticRegression_classifier accuracy percent:" ,
(nltk.classify.accuracy(LogisticRegression_classifier,
testing_set))* 100 )


Save_classifier = open(
"pickled_algos/LogisticRegression_classifier5k.pickle" ,
"wb" )
Pickle.dump(LogisticRegression_classifier, save_classifier)
Save_classifier.close()



LinearSVC_classifier = SklearnClassifier(LinearSVC())
LinearSVC_classifier.train(training_set)
Print( "LinearSVC_classifier accuracy percent:" ,
(nltk.classify.accuracy(LinearSVC_classifier,
testing_set))* 100 )


Save_classifier = open(
"pickled_algos/LinearSVC_classifier5k.pickle" , "wb" )
Pickle.dump(LinearSVC_classifier, save_classifier)
Save_classifier.close()



##NuSVC_classifier = SklearnClassifier(NuSVC())
##NuSVC_classifier.train(training_set)
##print("NuSVC_classifier accuracy percent:",
(nltk.classify.accuracy(NuSVC_classifier,
testing_set))*100)



SGDC_classifier = SklearnClassifier(SGDClassifier())
SGDC_classifier.train(training_set)
Print( "SGDClassifier accuracy percent:"
,nltk.classify.accuracy(SGDC_classifier, testing_set)* 100
)

Save_classifier = open(
"pickled_algos/SGDC_classifier5k.pickle" , "wb" )
Pickle.dump(SGDC_classifier, save_classifier)
Save_classifier.close()
```

Now you only need to run it once. If you wish, you can run it anytime, but now you are ready to create a sentiment analysis module. This is what we call `sentiment_mod.py` a file:

```
#File: sentiment_mod.py


Import nltk
 import random
 #from nltk.corpus import movie_reviews
from nltk.classify.scikitlearn import SklearnClassifier
 import pickle
 from sklearn.naive_bayes import MultinomialNB ,
BernoulliNB
 from sklearn.linear_model import LogisticRegression,
SGDClassifier
 from sklearn.svm import SVC, LinearSVC, NuSVC
 from nltk. Classify import ClassifierI
 from statistics import mode
 from nltk.tokenize import Word_tokenize

Class  VoteClassifier  (ClassifierI) :
    def  __init__  (self, *classifiers) :
        Self._classifiers = classifiers


    Def  classify  (self, features) :
        Votes = []
        For c in self._classifiers:
            v = c.classify(features)
            Votes.append(v)
        Return mode(votes)


    Def  confidence  (self, features) :
        Votes = []
        For c in self._classifiers:
            v = c.classify(features)
            Votes.append(v)


        Choice_votes = votes.count(mode(votes))
        Conf = choice_votes / len(votes)
        Return conf



Documents_f = open( "pickled_algos/documents.pickle" , "rb"
)
Documents = pickle.load(documents_f)
Documents_f.close()



Word_features5k_f = open(
"pickled_algos/word_features5k.pickle" , "rb" )
Word_features = pickle.load(word_features5k_f)
```

```
Word_features5k_f.close()



Def  find_features  (document) :
    Words = word_tokenize(document)
    Features = {}
    For w in word_features:
        Features[w] = (w in words)


    Return features



Featuresets_f
= open( "pickled_algos/featuresets.pickle" , "rb" )
Featuresets = pickle.load(featuresets_f)
Featuresets_f.close()


Random.shuffle(featuresets)
Print(len(featuresets))


Testing_set = featuresets[ 10000 :]
Training_set = featuresets[: 10000 ]



Open_file
= open( "pickled_algos/originalnaivebayes5k.pickle" , "rb"
)
Classifier = pickle.load(open_file)
Open_file.close()


Open_file
= open( "pickled_algos/MNB_classifier5k.pickle" , "rb" )
MNB_classifier = pickle.load(open_file)
Open_file.close()



Open_file
= open( "pickled_algos/BernoulliNB_classifier5k.pickle" ,
"rb" )
BernoulliNB_classifier = pickle.load(open_file)
Open_file.close()


Open_file
= open(
"pickled_algos/LogisticRegression_classifier5k.pickle" ,
"rb" )
LogisticRegression_classifier = pickle.load(open_file)
Open_file.close()


Open_file
= open( "pickled_algos/LinearSVC_classifier5k.pickle" ,
"rb" )
LinearSVC_classifier = pickle.load(open_file)
Open_file.close()
```

```
Open_file
= open( "pickled_algos/SGDC_classifier5k.pickle" , "rb" )
SGDC_classifier = pickle.load(open_file)
Open_file.close()



Voted_classifier = VoteClassifier(
                              Classifier,
                              LinearSVC_classifier,
                              MNB_classifier,
                              BernoulliNB_classifier,

LogisticRegression_classifier)
```

```
Def  sentiment  (text) :
    Feats = find_features(text)
    Return
voted_classifier.classify(feats),voted_classifier.confidenc
e(feats)
```

So here, in addition to the final function, there is nothing new, it is very simple. This function is the key to interacting with us from here. This function, which we call “emotion”, takes a parameter, text. Here, we use the `find_features` functions we have created to decompose these features. All we have to do now is to use our voting classifier to return the classification and return the confidence of the classification.

With this, we can now use this file, as well as the emotion function, as a module. The following is a sample script that uses this module:

```
Import sentiment_mod as s


Print(s.sentiment( "This movie was awesome! The acting was
great, plot was wonderful, and there were pythons...so
yea!" ))
Print(s.sentiment( "This movie was utter junk. There were
absolutely 0 pythons. I don't see what the point was at
all. Horrible movie, 0/10" ))
```

As expected, `python` the comments with the movie are obviously good, and no `python` movie is junk. Both have 100% confidence.

It took me about 5 seconds to import the module, because we saved the classifier and it might take 30 minutes without saving. Thanks to `pickle` your time, there will be big differences depending on your processor. If you keep going, I will say that you may want to see it too `joblib` .

Now that we have this great module, it works very easily, what can we do? I suggest that we go to Twitter for real-time sentiment analysis!

XX. NLTK Twitter sentiment analysis

Now that we have an sentiment analysis module, we can apply it to any text, but it's better to have short text, like Twitter! To this end, we will combine this tutorial with the Twitter Streaming API Tutorial.

The initial code for this tutorial is:

```
From tweepy import Stream
 from tweepy import OAuthHandler
 from tweepy.streaming import StreamListener



#consumer key, consumer secret, access token, access
secret.
ckey= "fsdfasdfsafsffa"
csecret= "asdfsadfsadfsadf"
atoken= "asdf-aassdfs"
asecret= "asdfsadfsdafsdafs"


Class  listener  (StreamListener) :


    Def  on_data  (self, data) :
        Print(data)
        Return ( True )


    Def  on_error  (self, status) :
        print status


Auth = OAuthHandler(ckey, csecret)
Auth.set_access_token(atoken, asecret)


twitterStream = Stream(auth, listener())
twitterStream.filter(track=[ "car" ])
```

This is enough to print `car` all the data of a streaming live tweet containing words . We can use `json` modules `json.loads(data)` to load data variables, and then we can reference specific ones `tweet` :

```
Tweet = all_data[ "text" ]
```

Since we have a tweet, we can easily pass it to our `sentiment_mod` module.

```
From tweepy import Stream
 from tweepy import OAuthHandler
 from tweepy.streaming import StreamListener
 import json
 import sentiment_mod as s


#consumer key, consumer secret, access token, access
secret.
ckey= "asdfsafsafsaf"
csecret= "
asdfasdfsadfsa " atoken= "asdfsadfsafsaf-asdfsaf"
asecret= "asdfsadfsadfsadfsadfsad"


From twitterapistuff import *

Class  listener  (StreamListener) :


    Def  on_data  (self, data) :


        All_data = json.loads(data)

        Tweet = all_data[ "text" ]
        Sentiment_value, confidence = s.sentiment(tweet)
        Print(tweet, sentiment_value, confidence)


        If confidence* 100 >= 80 :
            Output = open( "twitter-out.txt" , "a" )
            Output.write(sentiment_value)
            Output.write( '\n' )
            Output.close()


        Return  True

    Def  on_error  (self, status) :
        Print(status)
```

```
Auth = OAuthHandler(ckey, csecret)
Auth.set_access_token(atoken, asecret)


twitterStream = Stream(auth, listener())
twitterStream.filter(track=[ "happy" ])
```

In addition to this, we also save the results to the output file `twitter-out.txt` .

Next, what is the data analysis without the chart is complete? Let's combine another tutorial to draw a live streaming graph from sentiment analysis on the Twitter API.

XXI. Using NLTK to draw Twitter real-time sentiment analysis

Now that we have obtained real-time data from the Twitter Streaming API, why are there no activity diagrams showing emotional trends? To this end, we will combine this tutorial with the matplotlib drawing tutorial.

If you want to learn more about how the code works, see this tutorial. otherwise:

```
Import matplotlib.pyplot as plt
 import matplotlib.animation as animation
 from matplotlib import style
 import time

Style.use( "ggplot" )


Fig = plt.figure()
Ax1 = fig.add_subplot( 1 , 1 , 1 )


Def  animate  (i) :
    pullData = open( "twitter-out.txt" , "r" ).read()
    Lines = pullData.split( '\n' )

    Xar = []
    Yar = []
    x = 0
    y = 0

    For l in lines[- 200 :]:
        x += 1
        if  "pos"  in l:
            y += 1
        elif  "neg"  in l:
            y -= 1

        Xar.append(x)
        Yar.append(y)

    Ax1.clear()
    Ax1.plot(xar,yar)
Ani = animation.FuncAnimation(fig, animate, interval= 1000
)
Plt.show()
```

XXII. Stanford NER marker and named entity recognition

The Stanford NER marker provides an alternative to NLTK's Named Entity Recognition (NER) classifier. This marker is largely seen as a standard for named entity recognition, but because it uses advanced statistical learning algorithms, its computational overhead is greater than the options offered by NLTK.

One of the big advantages of the Stanford NER marker is that we have several different models to extract named entities. We can use any of the following:

- Three types of models for identifying locations, people, and organizations
- Four types of models for identifying locations, people, organizations, and miscellaneous entities
- Seven types of models, identifying location, people, organization, time, money, percentage and date

In order to continue, we need to download the model and `jar` files, because the NER classifier is written in Java. These are available free of charge from the Stanford Natural Language Processing Group . NTLK For the convenience of us, NLTK provides a wrapper for the Stanford marker, so we can use it in the best language (of course Python)!

`StanfordNERTagger` The parameters passed to the class include:

- The path of the classification model (the following three types of models are used)
- `jar` Path to the Stanford marker file
- Training data encoding (default is ASCII)

Here's how we set it up to mark sentences using three types of models:

```
# -*- coding: utf-8 -*-

From nltk.tag import StanfordNERTagger
 from nltk.tokenize import word_tokenize

St = StanfordNERTagger( '/usr/share/stanford-
ner/classifiers/english.all.3class.distsim.crf.ser.gz' ,
                        '/usr/share/stanford-ner/stanford-
ner.jar' ,
                       Encoding= 'utf-8' )

Text = 'While in France, Christine Lagarde discussed short-
term stimulus efforts in a recent interview with the Wall
Street Journal.'

Tokenized_text = word_tokenize(text)
Classified_text = st.tag(tokenized_text)

Print(classified_text)
```

Once we follow the word segmentation and classify the sentences, we will see that the marker produces the following list of tuples:

```
[('While', 'O'), ('in', 'O'), ('France', 'LOCATION'), (',',
'O'), ('Christine', 'PERSON') , ('Lagarde', 'PERSON'),
('discussed', 'O'), ('short-term', 'O'), ('stimulus', 'O'),
('efforts', 'O '), ('in', 'O'), ('a', 'O'), ('recent',
'O'), ('interview', 'O'), ('with', 'O '), ('the', 'O'),
('Wall', 'ORGANIZATION'), ('Street', 'ORGANIZATION'),
('Journal', 'ORGANIZATION'), ('.', 'O ')]
```

Great!Each tag uses `PERSON` , `LOCATION` , `ORGANIZATION` or `O` tags (using our three types of models). `O` Represents only the other, unnamed entities.

This list can now be used to test the annotated data, which we will cover in the next tutorial.

XXIII. Testing the accuracy of the NLTK and Stanford NER markers

We know how to use two different NER classifiers! But which one should we choose, NLTK or Stanford? Let's do some testing to find out the answer.

The first thing we need is some labeled reference data to test our NER classifier. One way to get this data is to find a large number of articles and mark each tag as a named entity (for example, people, organization, location) or other non-named entity. Then we can test our separate NER classifier with the correct label we know.

Unfortunately, this is very time consuming! The good news is that there is a manually annotated data set that is freely available with over 16,000 English sentences. There are also data sets in German, Spanish, French, Italian, Dutch, Polish, Portuguese and Russian!

This is an annotated sentence from the dataset:

```
Founding O
member O
Kojima I -PER
Minoru I -PER
Played O
guitar O
on O
Good I -MISC
Day I -MISC
, O
and O
```

```
Wardanceis the I -misc
Cover O
of O
A O
Song O
by O
UK the I -LOC
Post O
punk O
industrial O
band O
Killing I -ORG
Joke I -ORG
. O
```

Let’s read, split, and manipulate the data to make it a better format for testing.

```
Import nltk
from nltk.tag import StanfordNERTagger
from nltk.metrics.scores import accuracy


Raw_annotations = open( "/usr/share/wikigold.conll.txt"
).read()
Split_annotations = raw_annotations.split()

# Amend class annotations to reflect Stanford's NERTagger
for n,i in enumerate(split_annotations):
     if i == "I-PER" :
        Split_annotations[n] = "PERSON"
    if i == "I-ORG" :
        Split_annotations[n] = "ORGANIZATION"
    if i == "I-LOC" :
        Split_annotations[n] = "LOCATION"


# Group NE data into tuples
def  group  (lst, n) :
  for i in range( 0 , len(lst), n):
    Val = lst[i:i+n]
    If len(val) == n:
       yield tuple(val)

Reference_annotations = list(group(split_annotations, 2 ))
```

Ok, it looks good! However, we also need to paste the “clean” form of this data into our NER classifier. Let’s do it.

```
Pure_tokens = split_annotations[::2]
```

This reads in the data, splits it by whitespace, and then takes `split_annotations` a subset of everything in increments of two (starting from the zeroth element) . This produces a data set similar to the (smaller) example below:

```
['Founding', 'member', 'Kojima', 'Minoru', 'played',
'guitar', 'on', 'Good', 'Day', ',', 'and', 'Wardanceis', '
Cover', 'of', 'a', 'song', 'by', 'UK', 'post', 'punk',
'industrial', 'band', 'Killing', 'Joke', '.' ]
```

Let's go ahead and test the NLTK classifier:

```
Tagged_words = nltk.pos_tag(pure_tokens)
nltk_unformatted_prediction = nltk.ne_chunk(tagged_words)
```

Since the NLTK NER classifier generates trees (including POS tags), we need to do some extra data manipulation to get the proper form for testing.

```
#Convert prediction to multiline string and then to list
(includes pos tags)
Multiline_string =
nltk.chunk.tree2conllstr(nltk_unformatted_prediction)
Listed_pos_and_ne = multiline_string.split()

# Delete pos tags and rename
Del listed_pos_and_ne[ 1 :: 3 ]
Listed_ne = listed_pos_and_ne

# Amend class  annotations  for  consistency  with
reference_annotations
 for n,i in enumerate(listed_ne):
     if i == "B-PERSON" :
        Listed_ne[n] = "PERSON"
    if i == "I-PERSON" :
        Listed_ne[n] = "PERSON"
    if i == "B-ORGANIZATION" :
        Listed_ne[n] = "ORGANIZATION"
    if i == "I-ORGANIZATION" :
        Listed_ne[n] = "ORGANIZATION"
    if i == "B-LOCATION" :
        Listed_ne[n] = "LOCATION"
    if i == "I-LOCATION" :
        Listed_ne[n] = "LOCATION"
    if i == "B-GPE" :
        Listed_ne[n] = "LOCATION"
    if i == "I-GPE" :
        Listed_ne[n] = "LOCATION"


# Group prediction into tuples
Nltk_formatted_prediction = list(group(listed_ne, 2 ))
```

Now we can test the accuracy of NLTK.

```
Nltk_accuracy = accuracy(reference_annotations,
nltk_formatted_prediction) print(nltk_accuracy)
```

Wow, the accuracy rate `.8971` !

Let us now test the Stanford classifier. Because this classifier generates output in tuples, testing does not require more data manipulation.

```
St = StanfordNERTagger( '/usr/share/stanford-
ner/classifiers/english.all.3class.distsim.crf.ser.gz' ,
                        '/usr/share/stanford-ner/stanford-
ner.jar' ,
                       Encoding= 'utf-8' )
Stanford_prediction = st.tag(pure_tokens)
Stanford_accuracy = accuracy(reference_annotations,
stanford_prediction)
Print (stanford_accuracy)
```

`.9223` The accuracy rate! better!

If you want to draw this, here are some extra code. If you want to learn more about how this works, check out the matplotlib series:

```
Import numpy as np
Import matplotlib.pyplot as plt
from matplotlib import style

Style.use( 'fivethirtyeight' )


N = 1
ind = np.arange(N) # the x locations for the groups
width = 0.35  # the width of the bars

Fig, ax = plt.subplots()


Stanford_percentage = stanford_accuracy * 100
rects1 = ax.bar(ind, stanford_percentage, width, color= 'r'
)


Nltk_percentage = nltk_accuracy * 100
rects2 = ax.bar(ind+width, nltk_percentage, width, color=
'y' )

# add some text for labels, title and axes ticks
ax.set_xlabel( 'Classifier' )
Ax.set_ylabel( 'Accuracy (by percentage)' )
Ax.set_title( 'Accuracy by NER Classifier' )
Ax.set_xticks(ind+width)
Ax.set_xticklabels( ( '' ) )


Ax.legend( (rects1[ 0 ], rects2[ 0 ]), ( 'Stanford' ,
'NLTK' ), bbox_to_anchor=( 1.05 , 1 ), loc= 2 ,
borderaxespad= 0. )


Def  autolabel  (rects) :
    # attach some text labels
    for rect in rects:
        Height = rect.get_height()
        Ax.text(rect.get_x()+rect.get_width()/ 2. , 1.02
*height, '%10.2f' % float(height),
                Ha= 'center' , va= 'bottom' )

Autolabel(rects1)
Autolabel(rects2)


Plt.show()
```

XXIV. testing the speed of the NLTK and Stanford NER markers

We have tested the accuracy of our NER classifier, but there are more issues to consider when deciding which classifier to use. Let's test the speed!

We know that we are comparing the same thing, we will test it in the same article. Use this episode in NBC News:

```
House Speaker John Boehner became animated Tuesday over
the proposed Keystone Pipeline, castigating the Obama
administration for  not having the project.

Republican House Speaker John Boehner says there's "nothing
complex about the Keystone Pipeline,"  and  that  it 's
time  to build it .

"Complex? You think the Keystone Pipeline is complex?!"
Boehner responded to a questioner. "It's been under study
for five years! We build pipelines in America every day. Do
you realize that are 200,000 miles of pipelines in the
United States? "

At The Speaker Wentworth ON : "And at The only reason at
The President is's Involved in at The Keystone Pipeline IS
Because IT Crosses AN International's boundary the Listen,
WE CAN Build IT There's Nothing Complex the About at The
Keystone Pipeline - IT's Time to Build IT..."

Of Said Boehner at The President is NO excuse HAD AT the
this Point to  not give at The Pipeline at The Go-Ahead the
After  at The State Department Report Released A ON catalog
on Friday Indicating, at The Project Impact Would have have
A minimal ON  at The Environment.

Republicans have long pushed for construction of  the
project, which enjoys some measure of Democratic support as
well. The GOP is  considering conditioning an extension of
the debt limit on approval of  the project by Obama.

The White House, though, has said that  it has no timetable
for a final decision on  the project.
```

First, we perform the import and process the article through reading and word segmentation.

```
# -*- coding: utf-8 -*-


Import nltk
 import os
 Import numpy as np
Import matplotlib.pyplot as plt
 from matplotlib import style
 from nltk import pos_tag
 from nltk.tag import StanfordNERTagger
 from nltk.tokenize import word_tokenize

Style.use( 'fivethirtyeight' )


# Process text
def  process_text  (txt_file) :
    raw_text = open( "/usr/share/news_article.txt" ).read()
    Token_text = word_tokenize(raw_text)
    Return token_text
```

Great! Now let's write some functions to split our classification task. Because the NLTK NEG classifier requires a POS tag, we will add a POS tag to our NLTK function.

```
# Stanford NER tagger
def  stanford_tagger  (token_text) :
    st = StanfordNERTagger( '/usr/share/stanford-
ner/classifiers/english.all.3class.distsim.crf.ser.gz' ,
                             '/usr/share/stanford-ner
/stanford-ner.jar' ,
                            Encoding= 'utf-8' )
    Ne_tagged = st.tag(token_text)
    Return (ne_tagged)

# NLTK POS and NER
taggers def  nltk_tagger  (token_text) :
    Tagged_words = nltk.pos_tag(token_text)
    Ne_tagged = nltk.ne_chunk(tagged_words)
    Return (ne_tagged)
```

Each classifier needs to read the article and classify the named entities, so we wrap these functions in a larger function to make timing easier.

```
Def  stanford_main  () :
    Print(stanford_tagger(process_text(txt_file)))

Def  nltk_main  () :
    print(nltk_tagger(process_text(txt_file)))
```

When we call our program, we call these functions. We will `os.times()` wrap our `stanford_main()` sum `nltk_main()` function in the function call , taking the fourth index, which is the elapsed time. Then we will plot our results.

```
If __name__ == '__main__' :
    Stanford_t0 = os .times()[ 4 ]
    Stanford_main()
    Stanford_t1 = os .times()[ 4 ]
    Stanford_total_time = stanford_t1 - stanford_t0

    Nltk_t0 = os .times()[ 4 ]
    Nltk_main()
    Nltk_t1 = os .times()[ 4 ]
    Nltk_total_time = nltk_t1 - nltk_t0

    Time_plot(stanford_total_time, nltk_total_time)
```

For our drawing, we use the `time_plot()` function:

```
Def  time_plot  (stanford_total_time, nltk_total_time) :
    N = 1
    ind = np.arange(N) # the x locations for the groups
    width = 0.35  # the width of the bars
    Stanford_total_time = stanford_total_time
    Nltk_total_time = nltk_total_time
    Fig, ax = plt.subplots()
    Rects1 = ax.bar(ind, stanford_total_time, width, color=
'r' )
    Rects2 = ax.bar(ind+width, nltk_total_time, width,
color= 'y' )

    # Add text for labels, title and axes ticks
    ax.set_xlabel( 'Classifier' )
    Ax.set_ylabel( 'Time (in seconds)' )
    Ax.set_title( 'Speed by NER Classifier' )
    Ax.set_xticks(ind+width)
    Ax.set_xticklabels( ( '' ) )
    Ax.legend( (rects1[ 0 ], rects2[ 0 ]), ( 'Stanford' ,
'NLTK' ), bbox_to_anchor=( 1.05 , 1 ), loc= 2 ,
borderaxespad= 0. )


    Def  autolabel  (rects) :
        # attach some text labels
        for rect in rects:
            Height = rect.get_height()
            Ax.text(rect.get_x()+rect.get_width()/ 2. ,
1.02 *height, '%10.2f' % float(height),
                    Ha= 'center' , va= 'bottom' )


    Autolabel(rects1)
    Autolabel(rects2)
    Plt.show()
```

Wow, NLTK is as fast as lightning! It seems that Stanford is more accurate, but NLTK is faster. This is important information to know when balancing the precision of our preferences with the computing resources required.

But wait, there is still a problem. Our output is ugly! This is a small sample from Stanford University:

```
[( 'House' , 'ORGANIZATION' ), ( 'Speaker' , 'O' ), (
'John' , 'PERSON' ), ( 'Boehner' , 'PERSON' ), ( 'became' ,
'O' ) , ( 'animated' , 'O' ), ( 'Tuesday' , 'O' ), ( 'over'
, 'O' ), ( 'the' , 'O' ), ( 'proposed' , 'O' ) ,(
'Keystone' , 'ORGANIZATION' ), ('Pipeline' , 'ORGANIZATION'
), ( ',' , 'O' ), ( 'castigating' , 'O' ), ( 'the' , 'O' ),
( 'Obama' , 'PERSON' ), ( 'administration' , 'O' ), ( 'for'
, 'O' ), ( 'not' , 'O' ), ( 'having' , 'O' ), ( 'approved'
, 'O' ),( 'the' , 'O' ), ( 'project' ,'O' ), ( 'yet' , 'O'
), ( '.' , 'O' )
```

And NLTK:

```
( S  ( ORGANIZATION House/NNP) Speaker/NNP ( PERSON
John/NNP Boehner/NNP) became/VBD animated/VBN Tuesday/NNP
over/IN the/DT proposed/VBN ( PERSON Keystone/NNP
Pipeline/NNP) , /, Castigating/VBG the/DT ( ORGANIZATION
Obama/NNP) administration/NN for/IN not/RB having/VBG
approved/VBN the/DT project/NN yet/RB ./.
```

Let’s turn them into a readable form in the next tutorial.

XXV. Create a readable list of named entities using BIO tags

Now that we have completed the test, let’s turn our named entity into a good readable format.

Again, we will use the same news from NBC News:

```
House Speaker John Boehner became animated Tuesday over
the proposed Keystone Pipeline, castigating the Obama
administration for  not having the project.

Republican House Speaker John Boehner says there's "nothing
complex about the Keystone Pipeline,"  and  that  it 's
time  to build it .

"Complex? You think the Keystone Pipeline is complex?!"
Boehner responded to a questioner. "It's been under study
for five years! We build pipelines in America every day. Do
you realize that are 200,000 miles of pipelines in the
United States? "

At The Speaker Wentworth ON : "And at The only reason at
The President is's Involved in at The Keystone Pipeline IS
Because IT Crosses AN International's boundary the Listen,
WE CAN Build IT There's Nothing Complex the About at The
Keystone Pipeline - IT's Time to Build IT..."

Of Said Boehner at The President is NO excuse HAD AT the
this Point to  not give at The Pipeline at The Go-Ahead the
After  at The State Department Report Released A ON catalog
on Friday Indicating, at The Project Impact Would have have
A minimal ON  at The Environment.

Republicans have long pushed for construction of  the
project, which enjoys some measure of Democratic support as
well. The GOP is  considering conditioning an extension of
the debt limit on approval of  the project by Obama.
```

```
The White House, though, has said that  it has no timetable
for a final decision on  the project.
```

Our NTLK output is already a tree (just the last step), so let's take a look at our Stanford output. We will mark the tag with BIO, B for the beginning of the named entity, I for the internal, and O for the other. For example, if our sentence is yes `Barack Obama went to Greece today`, we should mark it as `Barack-B Obama-I went-O to-O Greece-B today-O`. To do this, we will write a series of conditions to check the labels of the current and previous `O` tags.

```
# -*- coding: utf-8 -*-


Import nltk
 import os
 Import numpy as np
Import matplotlib.pyplot as plt
 from matplotlib import style
 from nltk import pos_tag
 from nltk.tag import StanfordNERTagger
 from nltk.tokenize import word_tokenize
 from nltk.chunk import conlltags2tree
 from nltk.tree import Tree

Style.use( 'fivethirtyeight' )


# Process text
def  process_text  (txt_file) :
    raw_text = open( "/usr/share/news_article.txt" ).read()
    Token_text = word_tokenize(raw_text)
    Return token_text


# Stanford NER tagger
def  stanford_tagger  (token_text) :
    st = StanfordNERTagger( '/usr/share/stanford-
ner/classifiers/english.all.3class.distsim.crf.ser.gz' ,
                             '/usr/share/stanford-ner
/stanford-ner.jar' ,
                            Encoding= 'utf-8' )
    Ne_tagged = st.tag(token_text)
    Return (ne_tagged)

# NLTK POS and NER
taggers def  nltk_tagger  (token_text) :
    Tagged_words = nltk.pos_tag(token_text)
    Ne_tagged = nltk.ne_chunk(tagged_words)
    Return (ne_tagged)
```

```
# Tag tokens with standard NLP BIO tags
def  bio_tagger  (ne_tagged) :
        Bio_tagged = []
        Prev_tag = "O"
        for token, tag in ne_tagged:
             if tag == "O" : #O
                Bio_tagged.append((token, tag))
                Prev_tag = tag
                Continue
            if tag != "O"  and prev_tag == "O" : # Begin NE
                bio_tagged.append((token, "B-" +tag))
                Prev_tag = tag
            Elif prev_tag != "O"  and prev_tag == tag: #
Inside NE
                bio_tagged.append((token, "I-" +tag))
                Prev_tag = tag
            Elif prev_tag != "O"  and prev_tag != tag: #
Adjacent NE
                bio_tagged.append((token, "B-" +tag))
                Prev_tag = tag
        Return bio_tagged
```

Now we write the BIO-tagged tags to the tree, so they are in the same format as the NLTK output.

```
# Create tree
def  stanford_tree  (bio_tagged) :
    Tokens, ne_tags = zip(*bio_tagged)
    Pos_tags = [pos for token, pos in pos_tag(tokens)]

    Conlltags = [(token, pos, ne) for token, pos, ne in
zip(tokens, pos_tags, ne_tags)]
    Ne_tree = conlltags2tree(conlltags)
    Return ne_tree
```

Traverse and parse out all named entities:

```
# Parse named entities from tree
def  structure_ne  (ne_tree) :
    Ne = []
    For subtree in ne_tree:
         if type(subtree) == Tree: # If subtree is a noun
chunk, ie NE != "O"
            Ne_label = subtree.label()
            Ne_string = " " .join([token for token, pos in
subtree.leaves()])
            Ne.append((ne_string, ne_label))
    Return ne
```

In our call, we put all the additional functions together.

```
Def  stanford_main  () :

Print(structure_ne(stanford_tree(bio_tagger(stanford_tagger
(process_text(txt_file))))))
```

```
Def  nltk_main  () :

print(structure_ne(nltk_tagger(process_text(txt_file))))
```

Then call these functions:

```
If __name__ == '__main__' :
    Stanford_main()
    Nltk_main()
```

Here is a nice looking output from Stanford:

```
[('House', 'ORGANIZATION'), ('John Boehner', 'PERSON'),
('Keystone Pipeline', 'ORGANIZATION'), ('Obama', 'PERSON'),
('Republican House', ' ORGANIZATION'), ('John Boehner',
'PERSON'), ('Keystone Pipeline', 'ORGANIZATION'),
('Keystone Pipeline', 'ORGANIZATION'), ('Boehner',
'PERSON'), ('America ', 'LOCATION'), ('United States',
'LOCATION'), ('Keystone Pipeline', 'ORGANIZATION'),
('Keystone Pipeline', 'ORGANIZATION'), ('Boehner',
'PERSON'), ('State Department', 'ORGANIZATION'),
('Republicans', 'MISC'), ('Democratic', 'MISC'), ('GOP',
'MISC'), ('Obama', 'PERSON'), ('White House', 'LOCATION')]
```

And from NLTK:

```
[('House', 'ORGANIZATION'), ('John Boehner', 'PERSON'),
('Keystone Pipeline', 'PERSON'), ('Obama', 'ORGANIZATION'),
('Republican', 'ORGANIZATION '), ('House', 'ORGANIZATION'),
('John Boehner', 'PERSON'), ('Keystone Pipeline',
'ORGANIZATION'), ('Keystone Pipeline', 'ORGANIZATION'),
('Boehner' , 'PERSON'), ('America', 'GPE'), ('United
States', 'GPE'), ('Keystone Pipeline', 'ORGANIZATION'),
('Listen', 'PERSON'), (' Keystone', 'ORGANIZATION'),
('Boehner', 'PERSON'), ('State Department',
'ORGANIZATION'), ('Democratic', 'ORGANIZATION'), ('GOP',
'ORGANIZATION'), ('Obama', 'PERSON'), ('White House',
'FACILITY')]
```

Work Hard. God Bless.

. . .

Related Stories from DDI:

Deep Learning Explained in 7 Steps—Data Driven Investor

Self-driving cars, Alexa, medical imaging—gadgets are getting super smart around us with the help of deep learning...

www.datadriveninvestor.com

8 Skills You Need to Become a Data Scientist—Data Driven Investor

Numbers do not scare you? There is nothing more satisfying than a beautiful excel sheet? You speak several languages...

www.datadriveninvestor.com