Tokenization with nltk

The NLTK module is a massive tool kit, aimed at helping you with the entire Natural Language Processing (NLP) methodology. NLTK will aid you with everything from splitting sentences from paragraphs, splitting up words, recognizing the part of speech of those words, highlighting the main subjects, and then even with helping your machine to understand what the text is all about. In this series, we're going to tackle the field of opinion mining, or sentiment analysis.

Now that you have all the things that you need, let's knock out some quick vocabulary:

* Corpus - Body of text, singular. Corpora is the plural of this. Example: A collection of medical journals.
* Lexicon - Words and their meanings. Example: English dictionary. Consider, however, that various fields will have different lexicons. For example: To a financial investor, the first meaning for the word "Bull" is someone who is confident about the market, as compared to the common English lexicon, where the first meaning for the word "Bull" is an animal. As such, there is a special lexicon for financial investors, doctors, children, mechanics, and so on.
* Token - Each "entity" that is a part of whatever was split up based on rules. For examples, each word is a token when a sentence is "tokenized" into words. Each sentence can also be a token, if you tokenized the sentences out of a paragraph.

Ex: 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))

o/p: ['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(word\_tokenize(EXAMPLE\_TEXT))

o/p: ['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', '.']

Stop words with NLTK

The idea of Natural Language Processing is to do some form of analysis, or processing, where the machine can understand, at least to some level, what the text means, says, or implies.

This is an obviously massive challenge, but there are steps to doing it that anyone can follow. The main idea, however, is that computers simply do not, and will not, ever understand words directly. Humans don't either \*shocker\*. In humans, memory is broken down into electrical signals in the brain, in the form of neural groups that fire in patterns. There is a lot about the brain that remains unknown, but, the more we break down the human brain to the basic elements, we find out basic the elements really are. Well, it turns out computers store information in a very similar way! We need a way to get as close to that as possible if we're going to mimic how humans read and understand text. Generally, computers use numbers for everything, but we often see directly in programming where we use binary signals (True or False, which directly translate to 1 or 0, which originates directly from either the presence of an electrical signal (True, 1), or not (False, 0)). To do this, we need a way to convert words to values, in numbers, or signal patterns. The process of converting data to something a computer can understand is referred to as "pre-processing." One of the major forms of pre-processing is going to be filtering out useless data. In natural language processing, useless words (data), are referred to as stop words.

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

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

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

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

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

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

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

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

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

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

filtered\_sentence = []

for w in word\_tokens:

if w not in stop\_words:

filtered\_sentence.append(w)

print(word\_tokens)

print(filtered\_sentence)

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

## Stemming words with NLTK

The idea of stemming is a sort of normalizing method. Many variations of words carry the same meaning, other than when tense is involved.

The reason why we stem is to shorten the lookup, and normalize sentences.

Consider:

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

This sentence means the same thing. in the car is the same. I was is the same. the ing denotes a clear past-tense in both cases, so is it truly necessary to differentiate between ride and riding, in the case of just trying to figure out the meaning of what this past-tense activity was?

No, not really.

This is just one minor example, but imagine every word in the English language, every possible tense and affix you can put on a word. Having individual dictionary entries per version would be highly redundant and inefficient, especially since, once we convert to numbers, the "value" is going to be identical.

One of the most popular stemming algorithms is the Porter stemmer, which has been around since 1979.

First, we're going to grab and define our stemmer:

from nltk.stem import PorterStemmer

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

ps = PorterStemmer()

Now, let's choose some words with a similar stem, like:

example\_words = ["python","pythoner","pythoning","pythoned","pythonly"]

Next, we can easily stem by doing something like:

for w in example\_words:

print(ps.stem(w))

Our output:

python

python

python

python

pythonli

Now let's try stemming a typical sentence, rather than 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))

Now our result is:

It

is

import

to

by

veri

pythonli

while

you

are

python

with

python

.

All

python

have

python

poorli

at

least

onc

.

## Part of Speech Tagging with NLTK

One of the more powerful aspects of the NLTK module is the Part of Speech tagging that it can do for you. This means labeling words in a sentence as nouns, adjectives, verbs...etc. Even more impressive, it also labels by tense, and more. Here's a list of the tags, what they mean, 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-3d take

VBZ verb, 3rd person sing. present takes

WDT wh-determiner which

WP wh-pronoun who, what

WP$ possessive wh-pronoun whose

WRB wh-abverb where, when

How might we use this? While we're at it, we're going to cover a new sentence tokenizer, called the PunktSentenceTokenizer. This tokenizer is capable of unsupervised machine learning, so you can actually train it on any body of text that you use. First, let's get some imports out of the way that we're going to use:

import nltk

from nltk.corpus import state\_union

from nltk.tokenize import PunktSentenceTokenizer

Now, let's create our training and testing data:

train\_text = state\_union.raw("2005-GWBush.txt")

sample\_text = state\_union.raw("2006-GWBush.txt")

One is a State of the Union address from 2005, and the other is from 2006 from past President George W. Bush.

Next, we can train the Punkt tokenizer like:

custom\_sent\_tokenizer = PunktSentenceTokenizer(train\_text)

Then we can actually tokenize, using:

tokenized = custom\_sent\_tokenizer.tokenize(sample\_text)

Now we can finish up this part of speech tagging script by creating a function that will run through and tag all of the parts of speech per sentence like so:

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 list of tuples, where the first element in the tuple is the word, and the second is the 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'),**

## Chunking with NLTK

Now that we know the parts of speech, we can do what is called chunking, and group words into hopefully meaningful chunks. One of the main goals of chunking is to group into what are known as "noun phrases." These are phrases of one or more words that contain a noun, maybe some descriptive words, maybe a verb, and maybe something like an adverb. The idea is to group nouns with the words that are in relation to them.

In order to chunk, we combine the part of speech tags with [regular expressions](https://pythonprogramming.net/regular-expressions-regex-tutorial-python-3/). Mainly from regular expressions, we are going to utilize the following:

+ = match 1 or more

? = match 0 or 1 repetitions.

\* = match 0 or MORE repetitions

. = Any character except a new line

See the tutorial linked above if you need help with regular expressions. The last things to note is that the part of speech tags are denoted with the "<" and ">" and we can also place regular expressions within the tags themselves, so account for things like "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)

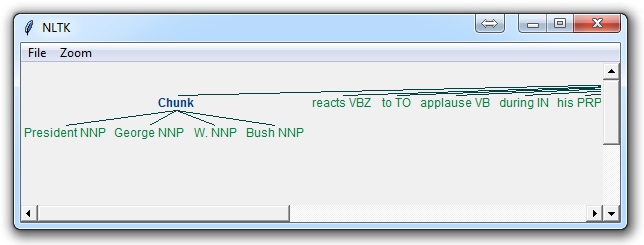
chunked.draw()

except Exception as e:

print(str(e))

process\_content()

The result of this is something like:



The main line here in question is:

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

This line, broken down:

<RB.?>\* = "0 or more of any tense of adverb," followed by:

<VB.?>\* = "0 or more of any tense of verb," followed by:

<NNP>+ = "One or more proper nouns," followed by

<NN>? = "zero or one singular noun."

Try playing around with combinations to group various instances until you feel comfortable with chunking.

Not covered in the video, but also a reasonable task is to actually access the chunks specifically. This is something rarely talked about, but can be an essential step depending on what you're doing. Say you print the chunks out, you are going to see output like:

(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, that helps us visually, but what if we want to access this data via our program? Well, what is happening here is our "chunked" variable is an NLTK tree. Each "chunk" and "non chunk" is a "subtree" of the tree. We can reference these by doing something like chunked.subtrees. We can then iterate through these subtrees like so:

for subtree in chunked.subtrees():

print(subtree)

Next, we might be only interested in getting just the chunks, 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're filtering to only show the subtrees with the label of "Chunk." Keep in mind, this isn't "Chunk" as in the NLTK chunk attribute... this is "Chunk" literally because that's the label we gave it here: chunkGram = r"""Chunk: {<RB.?>\*<VB.?>\*<NNP>+<NN>?}"""

Had we said instead something like chunkGram = r"""Pythons: {<RB.?>\*<VB.?>\*<NNP>+<NN>?}""", then we would filter by the label of "Pythons." The result here should be something like:

-

(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)

Full code for this would be:

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()

If you get particular enough, you may find that you may be better off if there was a way to chunk everything, except some stuff. This process is what is known as chinking, and that's what we're going to be covering next.

You may find that, after a lot of chunking, you have some words in your chunk you still do not want, but you have no idea how to get rid of them by chunking. You may find that chinking is your solution.

Chinking is a lot like chunking, it is basically a way for you to remove a chunk from a chunk. The chunk that you remove from your chunk is your chink.

The code is very similar, you just denote the chink, after the chunk, with }{ instead of the chunk's {}.

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)

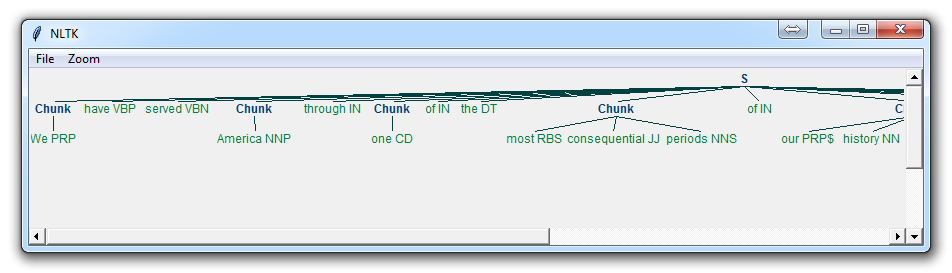
chunked.draw()

except Exception as e:

print(str(e))

process\_content()

With this, you are given something like:



Now, the main difference here is:

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

This means we're removing from the chink one or more verbs, prepositions, determiners, or the word 'to'.

## Named Entity Recognition with NLTK

One of the most major forms of chunking in natural language processing is called "Named Entity Recognition." The idea is to have the machine immediately be able to pull out "entities" like people, places, things, locations, monetary figures, and more.

This can be a bit of a challenge, but NLTK is this built in for us. There are two major options with NLTK's named entity recognition: either recognize all named entities, or recognize named entities as their respective type, like people, places, locations, etc.

Here's 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)

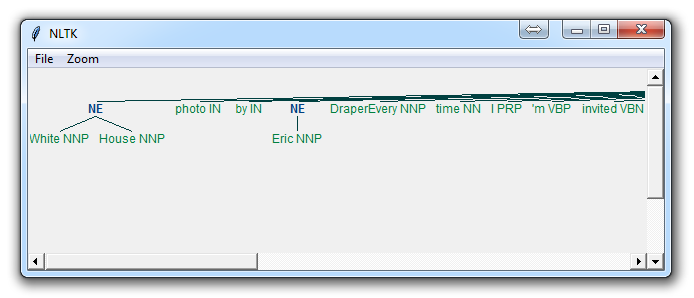
namedEnt.draw()

except Exception as e:

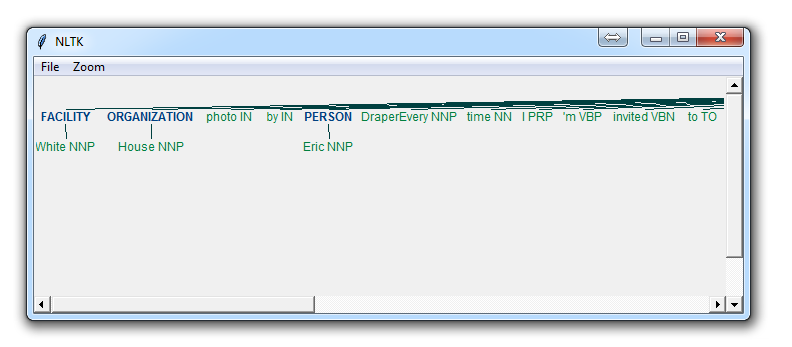
print(str(e))

process\_content()

Here, with the option of binary = True, this means either something is a named entity, or not. There will be no further detail. The result is:



If you set binary = False, then the result is:



Immediately, you can see a few things. When Binary is False, it picked up the same things, but wound up splitting up terms like White House into "White" and "House" as if they were different, whereas we could see in the binary = True option, the named entity recognition was correct to say White House was part of the same named entity.

Depending on your goals, you may use the binary option how you see fit. Here are the types of Named Entities that you can get if you have binary as false:

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 a m, 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 will probably find that you need to do a bit more work to get it just right, but this is pretty powerful right out of the box.

## Lemmatizing with NLTK

A very similar operation to stemming is called lemmatizing. The major difference between these is, as you saw earlier, stemming can often create non-existent words, whereas lemmas are actual words.

So, your root stem, meaning the word you end up with, is not something you can just look up in a dictionary, but you can look up a lemma.

Some times you will wind up with a very similar word, but sometimes, you will wind up with a completely different word. Let's see 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'))

o/p: cat

cactus

goose

rock

python

good

best

run

run

Here, we've got a bunch of examples of the lemma for the words that we use. The only major thing to note is that lemmatize takes a part of speech parameter, "pos." If not supplied, the default is "noun." This means that an attempt will be made to find the closest noun, which can create trouble for you. Keep this inmind if you use lemmatizing!

## The corpora with NLTK

In this part of the tutorial, I want us to take a moment to peak into the corpora we all downloaded! The NLTK corpus is a massive dump of all kinds of natural language data sets that are definitely worth taking a look at.

Almost all of the files in the NLTK corpus follow the same rules for accessing them by using the NLTK module, but nothing is magical about them. These files are plain text files for the most part, some are XML and some are other formats, but they are all accessible by you manually, or via the module and Python. Let's talk about viewing them manually.

Depending on your installation, your nltk\_data directory might be hiding in a multitude of locations. To figure out where it is, head to your Python directory, where the NLTK module is. If you do not know where that is, use the following code:

import nltk

print(nltk.\_\_file\_\_)

Run that, and the output will be the location of the NLTK module's \_\_init\_\_.py. Head into the NLTK directory, and then look for the data.py file.

The important blurb of 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 the nltk\_data. If you're on Windows, chances are it is in your appdata, in the local directory. To get there, you will want to open your file browser, go to the top, and type in %appdata%

Next click on roaming, and then find the nltk\_data directory. In there, you will have your corpora file. The full path is something like:   
C:\Users\yourname\AppData\Roaming\nltk\_data\corpora

Within here, you have all of the available corpora, including things like books, chat logs, movie reviews, and a whole lot more.

Now, we're going to talk about accessing these documents via NLTK. As you can see, these are mostly text documents, so you could just use normal Python code to open and read documents. That said, the NLTK module has a few nice methods for handling the corpus, so you may find it useful to use their methology. Here's an example of us opening the Gutenberg Bible, and reading 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])

## Wordnet with NLTK

[WordNet](https://wordnet.princeton.edu/) is a lexical database for the English language, which was created by Princeton, and is part of the NLTK corpus.

You can use WordNet alongside the NLTK module to find the meanings of words, synonyms, antonyms, and more. Let's cover some examples.

First, you're going to need to import wordnet:

from nltk.corpus import wordnet

Then, we're going to use the term "program" to find synsets like so:

syns = wordnet.synsets("program")

An example of a synset:

print(syns[0].name())

plan.n.01

Just the word:

print(syns[0].lemmas()[0].name())

plan

Definition of that first synset:

print(syns[0].definition())

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

Examples of the word in use:

print(syns[0].examples())

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

Next, how might we discern synonyms and antonyms to a word? The lemmas will be synonyms, and then you can use .antonyms to find the antonyms to the lemmas. As such, 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, we got many more synonyms than antonyms, since we just looked up the antonym for the first lemma, but you could easily balance this buy also doing the exact same process for the term "bad."

Next, we can also easily use WordNet to compare the similarity of two words and their tenses, by incorporating the [Wu and Palmer method](http://search.cpan.org/~tpederse/WordNet-Similarity-1.03/lib/WordNet/Similarity/wup.pm) for semantic related-ness.

Let's compare the noun of "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

## Text Classification with NLTK

Now that we're comfortable with NLTK, let's try to tackle text classification. The goal with text classification can be pretty broad. Maybe we're trying to classify text as about politics or the military. Maybe we're trying to classify it by the gender of the author who wrote it. A fairly popular text classification task is to identify a body of text as either spam or not spam, for things like email filters. In our case, we're going to try to create a sentiment analysis algorithm.

To do this, we're going to start by trying to use the movie reviews database that is part of the NLTK corpus. From there we'll try to use words as "features" which are a part of either a positive or negative movie review. The NLTK corpus movie\_reviews data set has the reviews, and they are labeled already as positive or negative. This means we can train and test with this data. First, let's wrangle 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 a moment to run this script, as the movie reviews dataset is somewhat large. Let's cover what is happening here.

After importing the data set we want, you see:

documents = [(list(movie\_reviews.words(fileid)), category)

for category in movie\_reviews.categories()

for fileid in movie\_reviews.fileids(category)]

Basically, in plain English, the above code is translated to: In each category (we have pos or neg), take all of the file IDs (each review has its own ID), then store the word\_tokenized version (a list of words) for the file ID, followed by the positive or negative label in one big list.

Next, we use random to shuffle our documents. This is because we're going to be training and testing. If we left them in order, chances are we'd train on all of the negatives, some positives, and then test only against positives. We don't want that, so we shuffle the data.

Then, just so you can see the data you are working with, we print out documents[1], which is a big list, where the first element is a list the words, and the 2nd element is the "pos" or "neg" label.

Next, we want to collect all words that we find, so we can have a massive list of typical words. From here, we can perform a frequency distribution, to then find out the most common words. As you will see, the most popular "words" are actually things like punctuation, "the," "a" and so on, but quickly we get to legitimate words. We intend to store a few thousand of the most popular words, so this shouldn't be a problem.

print(all\_words.most\_common(15))

The above gives you the 15 most common words. You can also find out how many occurences a word has by doing:

print(all\_words["stupid"])

## Converting words to Features with NLTK

In this tutorial, we're going to be building off the previous video and compiling feature lists of words from positive reviews and words from the negative reviews to hopefully see trends in specific types of words in positive or negative reviews.

To start, 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]

Mostly the same as before, only with now a new variable, word\_features, which contains the top 3,000 most common words. Next, we're going to build a quick function that will find these top 3,000 words in our positive and negative documents, marking their presence as either positive or negative:

def find\_features(document):

words = set(document)

features = {}

for w in word\_features:

features[w] = (w in words)

return features

Next, we can print one feature set like:

print((find\_features(movie\_reviews.words('neg/cv000\_29416.txt'))))

Then we can do this for all of our documents, saving the feature existence booleans and their respective positive or negative categories by doing:

featuresets = [(find\_features(rev), category) for (rev, category) in d

## Naive Bayes Classifier with NLTK

Now it is time to choose an algorithm, separate our data into training and testing sets, and press go! The algorithm that we're going to use first is the [Naive Bayes classifier](http://en.wikipedia.org/wiki/Naive_Bayes_classifier). This is a pretty popular algorithm used in text classification, so it is only fitting that we try it out first. Before we can train and test our algorithm, however, we need to go ahead and split up the data into a training set and a testing set.

You could train and test on the same dataset, but this would present you with some serious bias issues, so you should never train and test against the exact same data. To do this, since we've shuffled our data set, we'll assign the first 1,900 shuffled reviews, consisting of both positive and negative reviews, as the training set. Then, we can test against the last 100 to see how accurate we are.

This is called supervised machine learning, because we're showing the machine data, and telling it "hey, this data is positive," or "this data is negative." Then, after that training is done, we show the machine some new data and ask the computer, based on what we taught the computer before, what the computer thinks the category of the new data is.

We can split the data with:

# set that we'll train our classifier with

training\_set = featuresets[:1900]

# set that we'll test against.

testing\_set = featuresets[1900:]

Next, we can define, and train our classifier like:

classifier = nltk.NaiveBayesClassifier.train(training\_set)

First we just simply are invoking the Naive Bayes classifier, then we go ahead and use .train() to train it all in one line.

Easy enough, now it is trained. Next, we can test it:

print("Classifier accuracy percent:",(nltk.classify.accuracy(classifier, testing\_set))\*100)

Boom, you have your answer. In case you missed it, the reason why we can "test" the data is because we still have the correct answers. So, in testing, we show the computer the data without giving it the correct answer. If it guesses correctly what we know the answer to be, then the computer got it right. Given the shuffling that we've done, you and me might come up with varying accuracy, but you should see something from 60-75% on average.

Next, we can take it a step further to see what the most valuable words are when it comes to positive or negative reviews:

classifier.show\_most\_informative\_features(15)

This is going to vary again for each person, but you should see something like:

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 the ratio of occurences in negative to positive, or visa versa, for every word. So here, we can see that the term "insulting" appears 10.6 more times as often in negative reviews as it does in positive reviews. Ludicrous, 10.1.

Now, let's say you were totally content with your results, and you wanted to move forward, maybe using this classifier to predict things right now. It would be very impractical to train the classifier, and retrain it every time you needed to use it. As such, you can save the classifier using the pickle module. Let's do that next.

## Saving Classifiers with NLTK

Training classifiers and machine learning algorithms can take a very long time, especially if you're training against a larger data set. Ours is actually pretty small. Can you imagine having to train the classifier every time you wanted to fire it up and use it? What horror! Instead, what we can do is use the Pickle module to go ahead and serialize our classifier object, so that all we need to do is load that file in real quick.

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

save\_classifier = open("naivebayes.pickle","wb")

pickle.dump(classifier, save\_classifier)

save\_classifier.close()

This opens up a pickle file, preparing to write in bytes some data. Then, we use pickle.dump() to dump the data. The first parameter to pickle.dump() is what are you dumping, the second parameter is where are you dumping it.

After that, we close the file as we're supposed to, and that is that, we now have a pickled, or serialized, object saved in our script's directory!

Next, how would we go about opening and using this classifier? The .pickle file is a serialized object, all we need to do now is read it into memory, which will be about as quick as reading any other ordinary file. To do this:

classifier\_f = open("naivebayes.pickle", "rb")

classifier = pickle.load(classifier\_f)

classifier\_f.close()

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

Now, we can use this object, and we no longer need to train our classifier every time we wanted to use it to classify.

While this is all fine and dandy, we're probably not too content with the 60-75% accuracy we're getting. What about other classifiers? Turns out, there are many classifiers, but we need the scikit-learn (sklearn) module. Luckily for us, the people at NLTK recognized the value of incorporating the sklearn module into NLTK, and they have built us a little API to do it.

## Scikit-Learn Sklearn with NLTK

We've seen by now how easy it can be to use classifiers out of the box, and now we want to try some more! The best module for Python to do this with is the [Scikit-learn (sklearn) module](http://scikit-learn.org/stable/).

If you would like to learn more about the Scikit-learn Module, I have some tutorials on [machine learning with Scikit-Learn](https://pythonprogramming.net/machine-learning-python-sklearn-intro/).

Luckily for us, the people behind NLTK forsaw the value of incorporating the sklearn module into the NLTK classifier methodology. As such, they created the SklearnClassifier API of sorts. To use that, you just need to import it like:

from nltk.classify.scikitlearn import SklearnClassifier

From here, you can use just about any of the sklearn classifiers. For example, lets bring in a couple more variations of the Naive Bayes algorithm:

from sklearn.naive\_bayes import MultinomialNB,BernoulliNB

With this, how might we use them? It turns out, 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 is as simple as that. Let's bring in some more:

from sklearn.linear\_model import LogisticRegression,SGDClassifier

from sklearn.svm import SVC, LinearSVC, NuSVC

Now, all of our classifiers should look something like:

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)

The result of running this should give you something along the lines of:

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 SVC is wrong more often than it is right right out of the gate, so we should probably dump that one. But then what? The next thing we can try is to use all of these algorithms at once. An algo of algos! To do this, we can create another classifier, and make the result of that classifier based on what the other algorithms said. Sort of like a voting system, so we'll just need an odd number of algorithms.

## Combining Algorithms with NLTK

Now that we know how to use a bunch of algorithmic classifiers, like a child in the candy isle, told they can only pick one, we may find it difficult to choose just one classifier. The good news is, you don't have to! Combining classifier algorithms is is a common technique, done by creating a sort of voting system, where each algorithm gets one vote, and the classification that has the votes votes is the chosen one.

To do this, we want our new classifier to act like a typical NLTK classifier, with all of the methods. Simple enough, using object oriented programming, we can just be sure to inherit from the NLTK classifier class. To do this, we'll import it:

from nltk.classify import ClassifierI

from statistics import mode

We also import mode, as it will be our method for choosing the most popular vote.

Now, let's build our classifier class:

class VoteClassifier(ClassifierI):

def \_\_init\_\_(self, \*classifiers):

self.\_classifiers = classifiers

We're calling our class the VoteClassifier, and we're inheriting from NLTK's ClassifierI. Next, we're assigning the list of classifiers that are passed to our class to self.\_classifiers.

Next, we want to go ahead and create our own classify method. We want to call it classify, so that we can invoke .classify later on, like a traditional NLTK classifier would allow.

def classify(self, features):

votes = []

for c in self.\_classifiers:

v = c.classify(features)

votes.append(v)

return mode(votes)

Easy enough, all we're doing here is iterating through our list of classifier objects. Then, for each one, we ask it to classify based on the features. The classification is being treated as a vote. After we are done iterating, we then return the mode(votes), which is just returning the most popular vote.

This is all we really need, but I think it would be useful to have another parameter, confidence. Since we have algorithms voting, we can also tally the votes for and against the winning vote, and call this "confidence." For example, 3/5 votes for positive is weaker than 5/5 votes. As such, we can literally return the ratio of votes as a sort of confidence indicator. Here's 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 everything 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 here, we're running a few classification examples against text. All of 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

## Investigating bias with NLTK

The most major issue is that we have a fairly biased algorithm. You can test this yourself by commenting-out the shuffling of the documents, then training against the first 1900, and leaving the last 100 (all positive) reviews. Test, and you will find you have very poor accuracy.

Conversely, you can test against the first 100 data sets, all negative, and train against the following 1900. You will find very high accuracy here. This is a bad sign. It could mean a lot of things, and there are many options for us to fix it.

That said, the project I have in mind for us suggests we go ahead and use a different data set anyways, so we will do that. In the end, we will find this new data set still contains some bias, and that is that it picks up negative things more often. The reason for this is that negative reviews tend to be "more negative" than positive reviews are positive. Handling this can be done with some simple weighting, but it can also get complex fast. Maybe a tutorial for another day.

## Improving Training Data for sentiment analysis with NLTK

So now it is time to train on a new data set. Our goal is to do Twitter sentiment, so we're hoping for a data set that is a bit shorter per positive and negative statement. It just so happens that I have a data set of 5300+ positive and 5300+ negative movie reviews, which are much shorter. These should give us a bit more accuracy from the larger training set, as well as be more fitting for tweets from Twitter.

I have hosted both files here, you can find them by going to the [downloads for the short reviews](https://pythonprogramming.net/static/downloads/short_reviews/). Save these files as positive.txt and negative.txt.

Now, we can build our new data set in a very similar way as before. What needs to change?

We need a new methodology for creating our "documents" variable, and then we also need a new way to create the "all\_words" variable. No problem, really, here's how I did it:

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 finding function, mainly tokenizing by word in the document, since we didn't have a nifty .words() feature for our new sample. I also went ahead and increased 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 this, the rest is the same. Here's the full script just in case you or I missed something:

This process will take a while.. You may want to just go run some errands. It took me about 30-40 minutes to run it in full, and I am running an i7 3930k. For the typical processor in the year I am writing this (2015), it may be hours. This is a one and done process, however.

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 = True pos : 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

## Creating a module for Sentiment Analysis with NLTK

With this new dataset, and new classifier, we're ready to move forward. As you probably noticed, this new data set takes even longer to train against, since it's a larger set. As you've already been shown, we can actually save tons of time by pickling, or serializing, the trained classifiers, which are just objects.

You've already been shown how to use pickle to do this, so I encourage you to attempt to do it on your own. In case you need help, I will paste the full code to do that here...but seriously, do it yourself!

This process will take a while.. You may want to just go run some errands. It took me about 30-40 minutes to run it in full, and I am running an i7 3930k. For the typical processor in the year I am writing this (2015), it may be hours. This is a one and done process, however.

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 just need to run this one time. You can always run it again if you wanted, but now, you are ready to create the sentiment analysis module. Here's the file that we're going to call sentiment\_mod.py

#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.confidence(feats)

So here, there's really nothing new, besides the final function, which is quite simple. This function is the crux of what we will be interacting with from here on out. This function, which we're calling "sentiment," takes one parameter, which is text. From there, we break down the features with the find\_features function we created long ago. From there, now all we need to do is use our voted\_classifier to return not only the classification, but also the confidence in that classification.

With that, we can now use this file, and the sentiment function as a module. Here's an example script that might utilize the 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, the movie with pythons obviously did very well with reviewers, and the movie without any pythons was junk. Both of these were with 100% confidence as well.

It took me about 5 seconds to import the module, since we pickled the classifiers, as compared to the 30ish minutes it took without pickling. Yay for pickling. Your time will vary greatly depending on your processor. If you continue down this path, I will just throw out there that you may also want to look into joblib.

## Twitter Sentiment Analysis with NLTK

Now that we have a sentiment analysis module, we can apply it to just about any text, but preferrably short bits of text, like from Twitter! To do this, we're going to combine this tutorial with the [Twitter streaming API tutorial](https://pythonprogramming.net/twitter-api-streaming-tweets-python-tutorial/).

The initial code from that 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"])

That is enough to print out all of the data for the streaming live tweets that contain the term "car." We can use the json module to load the data var with json.loads(data), and then we can reference the tweet specifically with:

tweet = all\_data["text"]

Now that we have a tweet, we can easily pass this through 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"])

Along with that, we're also saving the results to an output file, twitter-out.txt.

Next, what data analysis would be complete without graphs? Let's combine yet another tutorial with this one to make a live streaming graph from the sentiment analysis on the Twitter API!

## Graphing Live Twitter Sentiment Analysis with NLTK with NLTK

Now that we have live data coming in from the Twitter streaming API, why not also have a live graph that shows the sentiment trend? To do this, we're going to combine this tutorial with the [live matplotlib graphing tutorial](https://pythonprogramming.net/python-matplotlib-live-updating-graphs/).

If you want to know more about how the code works, see that 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()

## Named Entity Recognition with Stanford NER Tagger

An alternative to NLTK's named entity recognition (NER) classifier is provided by the Stanford NER tagger. This tagger is largely seen as the standard in named entity recognition, but since it uses an advanced statistical learning algorithm it's more computationally expensive than the option provided by NLTK.

A big benefit of the Stanford NER tagger is that is provides us with a few different models for pulling out named entities. We can use any of the following:

* 3 class model for recognizing locations, persons, and organizations
* 4 class model for recognizing locations, persons, organizations, and miscellaneous entities
* 7 class model for recognizing locations, persons, organizations, times, money, percents, and dates

In order to move forward we'll need to download the models and a jar file, since the NER classifier is written in Java. These are available for free from the [Stanford Natural Language Processing Group](http://nlp.stanford.edu/software/CRF-NER.shtml#Download). Conveniently for us, NTLK provides a wrapper to the Stanford tagger so we can use it in the best language ever (ahem, Python)!

The parameters passed to the StanfordNERTagger class include:

1. Classification model path (3 class model used below)
2. Stanford tagger jar file path
3. Training data encoding (default of ASCII)

Here's how we set it up to tag a sentence with the 3 class model:

# -\*- 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've tokenized by word and classified the sentence, we see the tagger produces a list of tuples as follows:

[('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')]

Nice! Each token is tagged (using our 3 class model) with either 'PERSON', 'LOCATION', 'ORGANIZATION', or 'O'. The 'O' simply stands for other, i.e., non-named entities.

## Testing NLTK and Stanford NER Taggers for Accuracy

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

The first thing we'll need is some annotated reference data on which to test our NER classifiers. One way to get this data would be to find lots of articles and label each token as a type of named entity (e.g., person, organization, location) or other non-named entity. Then we could test our separate NER classifiers against the labels we know are correct.

Unfortunately, this would be really time consuming! Good thing there's a [manually annotated dataset](http://schwa.org/projects/resources/wiki/Wikiner) available for free with over 16,000 English sentences. There are also datasets available in German, Spanish, French, Italian, Dutch, Polish, Portuguese, and Russian!

Here's one 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 I-MISC

cover O

of O

a O

song O

by O

UK 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 so it's in 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, that looks good! But we'll also need the “clean” form of that data to stick into our NER classifiers. Let's make that happen too.

pure\_tokens = split\_annotations[::2]

This reads in the data, splits it by spacing, then subsets everything in split\_annotations by an increment of 2 (starting with the 0th element). This produces a dataset like the following (much smaller) example:

['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 produces trees (including POS tags), we'll need to do some additional data manipulation to get it in a 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, .8971 accurate!

Now let's test the Stanford classifier. Since this classifier produces output in tuples, testing doesn't 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)

Sheesh, .9223 accuracy! Even better!

If you'd like to visualize this, here's some extra code. Check out the [matplotlib](https://www.youtube.com/playlist?list=PLQVvvaa0QuDfefDfXb9Yf0la1fPDKluPF) series if you'd like to figure out more about how this works:

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()

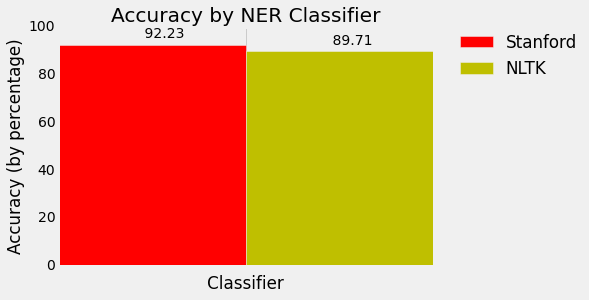
ax.text(rect.get\_x()+rect.get\_width()/2., 1.02\*height, '%10.2f' % float(height),

ha='center', va='bottom')

autolabel(rects1)

autolabel(rects2)

plt.show()



## Testing NLTK and Stanford NER Taggers for Speed

We've tested our NER classifiers for accuracy, but there's more we should consider in deciding which classifier to implement. Let's test for speed next!

Just so we know we're comparing apples to apples, we'll conduct our test on the same article. Let's go with this short piece from NBC news:

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

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 there are 200,000 miles of pipelines in the United States?"

The speaker went on: "And the only reason the president's involved in the Keystone Pipeline is because it crosses an international boundary. Listen, we can build it. There's nothing complex about the Keystone Pipeline -- it's time to build it."

Boehner said the president had no excuse at this point to not give the pipeline the go-ahead after the State Department released a report on Friday indicating the project would have a minimal impact on 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'll make our imports and process the article by reading and tokenizing.

# -\*- 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

Perfect! Now let's write some functions to separate out our classification tasks. We'll include POS tagging in our NLTK function since it's required for the NLTK NE classifier.

# 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'll wrap these functions in a larger function to make timing easy.

def stanford\_main():

print(stanford\_tagger(process\_text(txt\_file)))

def nltk\_main():

print(nltk\_tagger(process\_text(txt\_file)))

Let's call the functions when we call our program. We'll wrap our stanford\_main() and nltk\_main() functions in os.times() functions, taking the 4th index, elapsed time. Then we'll graph 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 graph, we'll be using the folloing 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()

Whoa, NLTK is lightning fast! It seems Stanford is more accurate, but NLTK is faster. This is great information to know when weighing our preferred level of precision alongside required computing resources.

But wait, there's still a problem. Our output is ugly! Here's a small sample from Stanford:

[('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 from 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

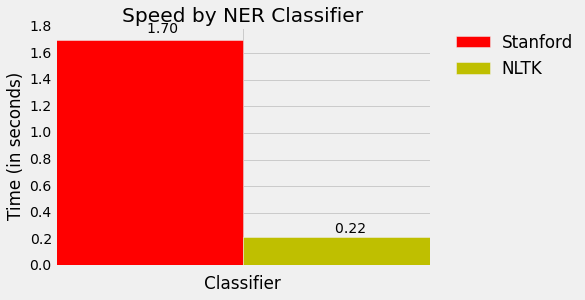
the/DT

project/NN

yet/RB

./.

Let's get these into nice readable forms in the next tutorial!



## Using BIO Tags to Create Readable Named Entity Lists

Now that we're done our testing, let's get our named entities in a nice readable format.

Again, we'll use the same short article from NBC news:

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

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 there are 200,000 miles of pipelines in the United States?"

The speaker went on: "And the only reason the president's involved in the Keystone Pipeline is because it crosses an international boundary. Listen, we can build it. There's nothing complex about the Keystone Pipeline -- it's time to build it."

Boehner said the president had no excuse at this point to not give the pipeline the go-ahead after the State Department released a report on Friday indicating the project would have a minimal impact on 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 in a tree (only requiring one last step), so let's get our Stanford output there as well. We'll start by BIO tagging the tokens, with B assigned to the beginning of named entities, I assigned to inside, and O assigned to other. For instance, if we have the sentence "Barack Obama went to Greece today", we should BIO tag it as "Barack-B Obama-I went-O to-O Greece-B today-O." In order to do this we'll write a series of conditionals to examine 'O' tags for current and previous tokens.

# -\*- 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'll write the BIO tagged tokens into trees, so they're in the same formate 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

Iterate through and parse out all the 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, i.e. NE != "O"

ne\_label = subtree.label()

ne\_string = " ".join([token for token, pos in subtree.leaves()])

ne.append((ne\_string, ne\_label))

return ne

We'll group all our additional functions together in our call:

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))))

And then call the functions:

if \_\_name\_\_ == '\_\_main\_\_':

stanford\_main()

nltk\_main()

Here's the 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')]

Nicely chunked together and readable. Sweet!