Setup

This guide was written in Python 3.6.

Libraries

We'll be working with the `re` library for regular expressions and nltk for natural language processing techniques, so make sure to install them! To install these libraries, enter the following commands into your terminal:

```
pip3 install re
pip3 install nltk
```

Other

Since we'll be working on textual analysis, we'll be using datasets that are already well established and widely used. To gain access to these datasets, enter the following command into your command line: (Note that this might take a few minutes!)

```
sudo python3 -m nltk.downloader all
```

Lastly, download the data we'll be working with in this example. You can find this in your folder as `negative_tweets` and `positive_tweets`.

Now you're all set to begin!

Background

What is NLP?

Natural Language Processing, or NLP, is an area of computer science that focuses on developing techniques to produce machine-driven analyses of text.

Why is Natural Language Processing Important?

NLP expands the sheer amount of data that can be used for insight. Since so much of the data we have available is in the form of text, this is extremely important to data science!

A specific common application of NLP is each time you use a language conversion tool. The techniques used to accurately convert text from one language to another very much falls under the umbrella of "natural language processing."

Why is NLP a "hard" problem?

Language is inherently ambiguous. Once person's interpretation of a sentence may very well differ from another person's interpretation. Because of this inability to consistently be clear, it's hard to have an NLP technique that works perfectly.

Glossary

Here is some common terminology that we'll encounter throughout the workshop:

Corpus: (Plural: Corpora) a collection of written texts that serve as our datasets.

nltk: (Natural Language Toolkit) the python module we'll be using repeatedly; it has a lot of useful built-in NLP techniques.

Token: a string of contiguous characters between two spaces, or between a space and punctuation marks. A token can also be an integer, real, or a number with a colon.

Sentiment Analysis

So you might be asking, what exactly is "sentiment analysis"?

Well, sentiment analysis involves building a system to collect and determine the emotional tone behind words. This is important because it allows you to gain an understanding of the attitudes, opinions and emotions of the people in your data.

At a high level, sentiment analysis involves Natural language processing and artificial intelligence by taking the actual text element, transforming it into a format that a machine can read, and using statistics to determine the actual sentiment.

Preparing the Data

To accomplish sentiment analysis computationally, we have to use techniques that will allow us to learn from data that's already been labeled.

So what's the first step? Formatting the data so that we can actually apply NLP techniques.

In [2]:
```python
import nltk

def format_sentence(sent):
    return({word: True for word in nltk.word_tokenize(sent)})

print(nltk.word_tokenize("The cat is very cute"))
```

```
['The', 'cat', 'is', 'very', 'cute']
```

Here, `format_sentence` changes a piece of text, in this case a tweet, into a dictionary of words mapped to True booleans. Though not obvious from this function alone, this will eventually allow us to train our prediction model by splitting the text into its tokens, i.e. *tokenizing* the text.

You'll learn about why this format is important in a later section.

Using the data on the github repo, we'll actually format the positively and negatively labeled data.

In [3]:
```python
pos = []
with open("./pos_tweets.txt") as f:
    for i in f:
        pos.append([format_sentence(i), 'pos'])
print(pos)
```

```
[[{'``': True, 'I': True, 'cheer': True, 'myself': True, 'up': True, 'whe
n': True, "'m": True, 'down': True, 'by': True, 'listening': True, 'to':
True, 'my': True, 'playlist': True, 'called': True, ',': True, 'Genius':
True, ':': True, 'Ballads': True, 'and': True, 'Cellos': True, '.': True,
'love': True, 'iPod': True, 'taste': True, 'of': True, 'music': True,
"''": True}, 'pos'], [{'``': True, 'just': True, 'watched': True, 'the':
True, 'movie': True, 'Wanted': True, '...': True, 'it': True, 'was': Tru
e, 'pretty': True, 'darn': True, 'good': True, '.': True, "''": True}, 'p
os'], [{'``': True, 'now': True, 'I': True, "'m": True, 'happy': True},
'pos'], [{'``': True, '--': True, 'plotting': True, 'like': True, 'i': Tr
ue, "'m": True, "mike..'game": True, 'plan': True, ':': True, 'pass': Tru
e, 'the': True, 'ball': True, 'to': True, 'lebron': True, 'AT': True, 'AL
L': True, 'TIMES': True, 'and': True, 'DONT': True, "FOUL'..certainly": T
rue, 'we': True, "'ll": True, 'win': True, 'haha..go': True, 'cavs': Tru
e, 'goooo': True, '!': True, "''": True}, 'pos'], [{'``': True, '@': Tru
e, 'mcdonalds': True, 'with': True, 'my': True, 'litto': True, 'sis': Tru
e, 'aka': True, 'cuzin': True, 'lol': True, 'cristyyyyy': True}, 'pos'],
[{'``': True, '@': True, 'PBnJen': True, ':': True, 'Thanks': True, 'fo
r': True, 'the': True, 'great': True, 'tour': True, 'and': True, 'makin
```

In [4]:
```python
neg = []
with open("./neg_tweets.txt") as f:
    for i in f:
        neg.append([format_sentence(i), 'neg'])
print(neg)
```

```
[[{'``': True, '@': True, 'iggigg': True, 'too': True, 'busy': True, 't
o': True, 'see': True, 'me': True, 'in': True, 'London': True, 'this': Tr
ue, 'evening': True, '.': True, 'What': True, 'is': True, 'a': True, 'bo
y': True, 'do': True, '?': True, "''": True}, 'neg'], [{'``': True, 'cav
s': True, 'lost': True, ',': True, 'and': True, 'I': True, 'got': True,
'this': True, 'sinking': True, 'feeling': True, 'we': True, 'are': True,
'going': True, 'to': True, 'lose': True, 'Lebron': True, 'in': True, '201
0': True, 'also': True, '...': True, 'why': True, 'must': True, 'my': Tru
e, 'home': True, 'city': True, 'SUCK': True, '?': True, 'Ah': True, 'wel
l': True, 'LETS': True, 'GO': True, 'BROWNS': True, '!': True, "''": Tru
e}, 'neg'], [{'``': True, 'the': True, 'closest': True, 'BGT': True, 'tou
r': True, 'is': True, 'Cardiff': True, 'or': True, 'London': True, 'dam':
True, 'it': True, 'why': True, 'doesnt': True, 'anybody': True, 'other':
True, 'than': True, 'Chuckle': True, 'Brothers': True, 'Westcountry': Tru
e, '?': True, "''": True}, 'neg'], [{'``': True, 'Why': True, 'do': True,
'other': True, 'pet': True, 'care': True, 'people': True, 'try': True, 't
o': True, 'run': True, 'others': True, 'out': True, 'of': True, 'busines
s': True, '?': True, 'Or': True, 'send': True, 'suspicious': True, 'e-mai
ls': True, 'fishing': True, 'for': True, 'info': True, "''": True}, 'ne
```

Training Data

Next, we'll split the labeled data we have into two pieces, one that can "train" data and the other to give us insight on how well our model is performing. The training data will inform our model on which features are most important.

In [5]:
```
training = pos[:int((.9)*len(pos))] + neg[:int((.9)*len(neg))]
```

Test Data

We won't use the test data until the very end of this section, but nevertheless, we save the last 10% of the data to check the accuracy of our model.

In [6]:
```
test = pos[int((.1)*len(pos)):] + neg[int((.1)*len(neg)):]

print(test)
```

```
[[{'``': True, '@': True, 'Dannymcfly': True, 'heyhey': True, 'and': Tru
e, 'they': True, 'love': True, 'youuu': True, '!': True, 'hahahaha': Tru
e, 'having': True, 'a': True, 'good': True, 'time': True, 'over': True,
'in': True, 'brazil': True, 'then': True, '?': True, "''": True}, 'pos'],
[{'``': True, '@': True, 'danzelikman': True, 'Here': True, "'s": True,
'to': True, 'hoping': True, 'you': True, 'come': True, 'home': True, 'wit
h': True, 'a': True, 'Las': True, 'Vegas': True, 'bailout.': True}, 'po
s'], [{'``': True, '@': True, 'DaRealSunisaKim': True, 'Thanks': True, 'f
or': True, 'the': True, 'Twitter': True, 'add': True, ',': True, 'Sunis
a': True, '!': True, 'I': True, 'got': True, 'to': True, 'meet': True, 'y
ou': True, 'once': True, 'at': True, 'a': True, 'HIN': True, 'show': Tru
e, 'here': True, 'in': True, 'DC': True, 'area': True, 'and': True, 'wer
e': True, 'sweetheart.': True}, 'pos'], [{'``': True, '@': True, 'DaveBo
s': True, 'its': True, 'okay': True, 'ive': True, 'done': True, 'it': Tru
e, 'once': True, 'didnt': True, 'but': True, 'Wooo': True, 'Stayed': Tru
e, 'up': True, 'longer': True, 'then': True, 'expected': True, 'lol': Tru
e}, 'pos'], [{'``': True, '@': True, 'DaveMatthewsB': True, 'New': True,
'album': True, 'is': True, 'fantastic': True, '!': True, 'See': True, 'yo
u': True, 'in': True, 'London': True, ',': True, 'ca': True, "n't": True,
```

Building a Classifier

All NLTK classifiers work with feature structures, which can be simple dictionaries mapping a feature name to a feature value. In this example, we've used a simple bag of words model where every word is a feature name with a value of True.

In [7]:
```
from nltk.classify import NaiveBayesClassifier

classifier = NaiveBayesClassifier.train(training)
```

To see which features informed our model the most, we can run this line of code:

In [8]:
```
classifier.show_most_informative_features()
```

```
Most Informative Features
                      no = True              neg : pos    =     20.6 : 1.0
                 awesome = True              pos : neg    =     18.7 : 1.0
                headache = True              neg : pos    =     18.0 : 1.0
               beautiful = True              pos : neg    =     14.2 : 1.0
                    love = True              pos : neg    =     14.2 : 1.0
                      Hi = True              pos : neg    =     12.7 : 1.0
                    glad = True              pos : neg    =      9.7 : 1.0
                   Thank = True              pos : neg    =      9.7 : 1.0
                     fan = True              pos : neg    =      9.7 : 1.0
                    lost = True              neg : pos    =      9.3 : 1.0
```

Classification

Just to see that our model works, let's try the classifier out with a positive example:

In [29]:
```
example1 = "I have no headache"

print(classifier.classify(format_sentence(example1)))
```

```
neg
```

Now for a negative example:

In [10]:
```
example2 = "this workshop is awful."

print(classifier.classify(format_sentence(example2)))
```

```
neg
```

In [11]:
```
example2 = "I have no headache"

print(classifier.classify(format_sentence(example2)))
```

```
neg
```

Accuracy

Now, there's no point in building a model if it doesn't work well. Luckily, once again, nltk comes to the rescue with a built in feature that allows us find the accuracy of our model.

In [12]:
```
from nltk.classify.util import accuracy
print(accuracy(classifier, test))
```

```
0.9562326869806094
```

Turns out it works decently well!

But it could be better! I think we can agree that the data is kind of messy - there are typos, abbreviations, grammatical errors of all sorts... So how do we handle that? Can we handle that?

Regular Expressions

A regular expression is a sequence of characters that define a string.

Simplest Form

The simplest form of a regular expression is a sequence of characters contained within **two backslashes**. For example, *python* would be

```
\python
```

Case Sensitivity

Regular Expressions are **case sensitive**, which means

```
\p and \P
```

are distinguishable from eachother. This means *python* and *Python* would have to be represented differently, as follows:

```
\python and \Python
```

We can check these are different by running:

In [13]:
```
import re
re1 = re.compile('python')
print(bool(re1.match('Python')))
```

```
False
```

Disjunctions

If you want a regular expression to represent both *python* and *Python*, however, you can use **brackets** or the **pipe** symbol as the disjunction of the two forms. For example,

```
[Pp]ython or \Python|python
```

could represent either *python* or *Python*. Likewise,

```
[0123456789]
```

would represent a single integer digit. The pipe symbols are typically used for interchangable strings, such as in the following example:

```
\dog|cat
```

Ranges

If we want a regular expression to express the disjunction of a range of characters, we can use a **dash**. For example, instead of the previous example, we can write

```
[0-9]
```

Similarly, we can represent all characters of the alphabet with

```
[a-z]
```

Exclusions

Brackets can also be used to represent what an expression **cannot** be if you combine it with the **caret** sign. For example, the expression

```
[^p]
```

represents any character, special characters included, but p.

Question Marks

Question marks can be used to represent the expressions containing zero or one instances of the previous character. For example,

```
<i>\colou?r
```

represents either *color* or *colour*. Question marks are often used in cases of plurality. For example,

```
<i>\computers?
```

can be either *computers* or *computer*. If you want to extend this to more than one character, you can put the simple sequence within parenthesis, like this:

```
\Feb(ruary)?
```

This would evaluate to either *February* or *Feb*.

Kleene Star

To represent the expressions containing zero or **more** instances of the previous character, we use an **asterisk** as the kleene star. To represent the set of strings containing *a, ab, abb, abbb, ...*, the following regular expression would be used:

```
\ab*
```

Wildcards

Wildcards are used to represent the possibility of any character and symbolized with a **period**. For example,

```
\beg.n
```

From this regular expression, the strings *begun, begin, began,* etc., can be generated.

Kleene+

To represent the expressions containing at **least** one or more instances of the previous character, we use a **plus** sign. To represent the set of strings containing *ab, abb, abbb, ...*, the following regular expression would be used:

```
\ab+
```

Word Tagging and Models

Given any sentence, you can classify each word as a noun, verb, conjunction, or any other class of words. When there are hundreds of thousands of sentences, even millions, this is obviously a large and tedious task. But it's not one that can't be solved computationally.

NLTK Parts of Speech Tagger

NLTK is a package in python that provides libraries for different text processing techniques, such as classification, tokenization, stemming, parsing, but important to this example, tagging.

In [14]:
```
import nltk

text = nltk.word_tokenize("Python is an awesome language!")
print(text)
nltk.pos_tag(text)
```

```
['Python', 'is', 'an', 'awesome', 'language', '!']
```

Out[14]:
```
[('Python', 'NNP'),
 ('is', 'VBZ'),
 ('an', 'DT'),
 ('awesome', 'JJ'),
 ('language', 'NN'),
 ('!', '.')]
```

Not sure what DT, JJ, or any other tag is? Just try this in your python shell:

In [15]:
```
nltk.help.upenn_tagset('EP')
```

```
No matching tags found.
```

Ambiguity

But what if a word can be tagged as more than one part of speech? For example, the word "sink." Depending on the content of the sentence, it could either be a noun or a verb.

Furthermore, what if a piece of text demonstrates a rhetorical device like sarcasm or irony? Clearly this can mislead the sentiment analyzer to misclassify a regular expression.

Unigram Models

Remember our bag of words model from earlier? One of its characteristics was that it didn't take the ordering of the words into account - that's why we were able to use dictionaries to map each words to True values.

With that said, unigram models are models where the order doesn't make a difference in our model. You might be wondering why we care about unigram models since they seem to be so simple, but don't let their simplicity fool you - they're a foundational block for a lot of more advanced techniques in NLP.

In [16]:

```
from nltk.corpus import brown

brown_tagged_sents = brown.tagged_sents(categories='news')
brown_sents = brown.sents(categories='news')
unigram_tagger = nltk.UnigramTagger(brown_tagged_sents)
unigram_tagger.tag(brown_sents[2007])
```

Out[16]:

```
[('Various', 'JJ'),
 ('of', 'IN'),
 ('the', 'AT'),
 ('apartments', 'NNS'),
 ('are', 'BER'),
 ('of', 'IN'),
 ('the', 'AT'),
 ('terrace', 'NN'),
 ('type', 'NN'),
 (',', ','),
 ('being', 'BEG'),
 ('on', 'IN'),
 ('the', 'AT'),
 ('ground', 'NN'),
 ('floor', 'NN'),
 ('so', 'QL'),
 ('that', 'CS'),
 ('entrance', 'NN'),
 ('is', 'BEZ'),
 ('direct', 'JJ'),
 ('.', '.')]
```

Bigram Models

Here, ordering does matter.

Type *Markdown* and LaTeX: α^2

In [17]:
```
bigram_tagger = nltk.BigramTagger(brown_tagged_sents)
bigram_tagger.tag(brown_sents[2007])
```

Out[17]:
```
[('Various', 'JJ'),
 ('of', 'IN'),
 ('the', 'AT'),
 ('apartments', 'NNS'),
 ('are', 'BER'),
 ('of', 'IN'),
 ('the', 'AT'),
 ('terrace', 'NN'),
 ('type', 'NN'),
 (',', ','),
 ('being', 'BEG'),
 ('on', 'IN'),
 ('the', 'AT'),
 ('ground', 'NN'),
 ('floor', 'NN'),
 ('so', 'CS'),
 ('that', 'CS'),
 ('entrance', 'NN'),
 ('is', 'BEZ'),
 ('direct', 'JJ'),
 ('.', '.')]
```

Notice the changes from the last time we tagged the words of this same sentence.

Normalizing Text

The best data is data that's consistent - textual data usually isn't. But we can make it that way by normalizing it. To do this, we can do a number of things.

At the very least, we can make all the text so that it's all in lowercase. You may have already done this before:

Given a piece of text,

In [18]:
```
raw = "OMG, Natural Language Processing is SO cool and I'm really enjoying th
tokens = nltk.word_tokenize(raw)
tokens = [i.lower() for i in tokens]
print(tokens)
```

```
['omg', ',', 'natural', 'language', 'processing', 'is', 'so', 'cool', 'an
d', 'i', "'m", 'really', 'enjoying', 'this', 'workshop', '!']
```

Stemming

But we can do more!

What is Stemming?

Stemming is the process of converting the words of a sentence to its non-changing portions. In the example of amusing, amusement, and amused above, the stem would be amus.

Types of Stemmers

You're probably wondering how do I convert a series of words to its stems. Luckily, NLTK has a few built-in and established stemmers available for you to use! They work slightly differently since they follow different rules - which you use depends on whatever you happen to be working on.

First, let's try the Lancaster Stemmer:

In [19]:
```
lancaster = nltk.LancasterStemmer()
stems = [lancaster.stem(i) for i in tokens]
```

This should have the output:

In [20]:
```
print(stems)
```

```
['omg', ',', 'nat', 'langu', 'process', 'is', 'so', 'cool', 'and', 'i',
"'m", 'real', 'enjoy', 'thi', 'workshop', '!']
```

Secondly, we try the Porter Stemmer:

In [21]:
```
porter = nltk.PorterStemmer()
stem = [porter.stem(i) for i in tokens]
```

Notice how "natural" maps to "natur" instead of "nat" and "really" maps to "realli" instead of "real" in the last stemmer.

In [22]:
```
print(stem)
```

```
['omg', ',', 'natur', 'languag', 'process', 'is', 'so', 'cool', 'and', 'i',
"'m", 'realli', 'enjoy', 'thi', 'workshop', '!']
```

Lemmatization

What is Lemmatization?

Lemmatization is the process of converting the words of a sentence to its dictionary form. For example, given the words amusement, amusing, and amused, the lemma for each and all would be amuse.

WordNetLemmatizer

Once again, NLTK is awesome and has a built in lemmatizer for us to use:

Type *Markdown* and LaTeX: α^2

In [23]:
```python
from nltk import WordNetLemmatizer

lemma = nltk.WordNetLemmatizer()
text = "Women in technology are amazing at coding"
ex = [i.lower() for i in text.split()]

lemmas = [lemma.lemmatize(i) for i in ex]
print(lemmas)
```

```
['woman', 'in', 'technology', 'are', 'amazing', 'at', 'coding']
```

Notice that women is changed to "woman"

Final Words

Going back to our original sentiment analysis, we could have improved our model in a lot of ways by applying some of techniques we just went through. The twitter data is seemingly messy and inconsistent, so if we really wanted to get a highly accurate model, we could have done some preprocessing on the tweets to clean it up.

Secondly, the way in which we built our classifier could have been improved. Our feature extraction was relatively simple and could have been improved by using a bigram model rather than the bag of words model. We could have also fixed our Bayes Classifier so that it only took the most frequent words into considerations.

Resources

Natural Language Processing With Python (http://bit.ly/nlp-w-python)
Regular Expressions Cookbook (http://bit.ly/regular-expressions-cb)

In [ ]:
```

```