NLP (Natural Language Processing) with Python

This is the notebook that goes along with the NLP video lecture!

In this lecture we will discuss a higher level overview of the basics of Natural Language Processing, which basically consists of combining machine learning techniques with text, and using math and statistics to get that text in a format that the machine learning algorithms can understand!

Once you've completed this lecture you'll have a project using some Yelp Text Data!

Requirements: You will need to have NLTK installed, along with downloading the corpus for stopwords. To download everything with a conda installation, run the cell below. Or reference the full video lecture

In [7]:
```
import nltk
```

In [8]:
```
nltk.download_shell()
```

```
NLTK Downloader
---------------------------------------------------------------------------
    d) Download   l) List    u) Update   c) Config   h) Help   q) Quit
---------------------------------------------------------------------------
Downloader> l

Packages:
  [ ] abc..................... Australian Broadcasting Commission 2006
  [ ] alpino.................. Alpino Dutch Treebank
  [ ] averaged_perceptron_tagger Averaged Perceptron Tagger
  [ ] averaged_perceptron_tagger_ru Averaged Perceptron Tagger (Russian)
  [ ] basque_grammars..... Grammars for Basque
  [ ] bcp47................... BCP-47 Language Tags
  [ ] biocreative_ppi..... BioCreAtIvE (Critical Assessment of Information
                           Extraction Systems in Biology)
  [ ] bllip_wsj_no_aux.... BLLIP Parser: WSJ Model
  [ ] book_grammars....... Grammars from NLTK Book
  [ ] brown................... Brown Corpus
  [ ] brown_tei........... Brown Corpus (TEI XML Version)
  [ ] cess_cat............ CESS-CAT Treebank
  [ ] cess_esp............ CESS-ESP Treebank
  [ ] chat80.............. Chat-80 Data Files
  [ ] city_database....... City Database
  [ ] cmudict............. The Carnegie Mellon Pronouncing Dictionary (0.6)
  [ ] comparative_sentences Comparative Sentence Dataset
  [ ] comtrans............ ComTrans Corpus Sample
  [ ] conll2000........... CONLL 2000 Chunking Corpus
Hit Enter to continue:
  [ ] conll2002........... CONLL 2002 Named Entity Recognition Corpus
  [ ] conll2007........... Dependency Treebanks from CoNLL 2007 (Catalan
                           and Basque Subset)
  [ ] crubadan............ Crubadan Corpus
  [ ] dependency_treebank. Dependency Parsed Treebank
  [ ] dolch............... Dolch Word List
  [ ] europarl_raw........ Sample European Parliament Proceedings Parallel
                           Corpus
  [ ] extended_omw........ Extended Open Multilingual WordNet
  [ ] floresta............ Portuguese Treebank
  [ ] framenet_v15........ FrameNet 1.5
  [ ] framenet_v17........ FrameNet 1.7
  [ ] gazetteers.......... Gazeteer Lists
  [ ] genesis............. Genesis Corpus
  [ ] gutenberg........... Project Gutenberg Selections
  [ ] ieer................ NIST IE-ER DATA SAMPLE
  [ ] inaugural........... C-Span Inaugural Address Corpus
  [ ] indian.............. Indian Language POS-Tagged Corpus
  [ ] jeita............... JEITA Public Morphologically Tagged Corpus (in
                           ChaSen format)
  [ ] kimmo............... PC-KIMMO Data Files
Hit Enter to continue:
  [ ] knbc................ KNB Corpus (Annotated blog corpus)
  [ ] large_grammars...... Large context-free and feature-based grammars
                           for parser comparison
  [ ] lin_thesaurus....... Lin's Dependency Thesaurus
  [ ] mac_morpho.......... MAC-MORPHO: Brazilian Portuguese news text with
                           part-of-speech tags
  [ ] machado............. Machado de Assis -- Obra Completa
  [ ] masc_tagged......... MASC Tagged Corpus
  [ ] maxent_ne_chunker... ACE Named Entity Chunker (Maximum entropy)
```

```
  [ ] maxent_treebank_pos_tagger Treebank Part of Speech Tagger (Maximum entropy)
  [ ] moses_sample........ Moses Sample Models
  [ ] movie_reviews....... Sentiment Polarity Dataset Version 2.0
  [ ] mte_teip5........... MULTEXT-East 1984 annotated corpus 4.0
  [ ] mwa_ppdb............ The monolingual word aligner (Sultan et al.
                           2015) subset of the Paraphrase Database.
  [ ] names............... Names Corpus, Version 1.3 (1994-03-29)
  [ ] nombank.1.0......... NomBank Corpus 1.0
  [ ] nonbreaking_prefixes Non-Breaking Prefixes (Moses Decoder)
  [ ] nps_chat............ NPS Chat
  [ ] omw-1.4............. Open Multilingual Wordnet
  [ ] omw................. Open Multilingual Wordnet
Hit Enter to continue:
  [ ] opinion_lexicon..... Opinion Lexicon
  [ ] panlex_swadesh...... PanLex Swadesh Corpora
  [ ] paradigms........... Paradigm Corpus
  [ ] pe08................ Cross-Framework and Cross-Domain Parser
                           Evaluation Shared Task
  [ ] perluniprops........ perluniprops: Index of Unicode Version 7.0.0
                           character properties in Perl
  [ ] pil................. The Patient Information Leaflet (PIL) Corpus
  [ ] pl196x.............. Polish language of the XX century sixties
  [ ] porter_test......... Porter Stemmer Test Files
  [ ] ppattach............ Prepositional Phrase Attachment Corpus
  [ ] problem_reports..... Problem Report Corpus
  [ ] product_reviews_1... Product Reviews (5 Products)
  [ ] product_reviews_2... Product Reviews (9 Products)
  [ ] propbank............ Proposition Bank Corpus 1.0
  [ ] pros_cons........... Pros and Cons
  [ ] ptb................. Penn Treebank
  [ ] punkt............... Punkt Tokenizer Models
  [ ] qc.................. Experimental Data for Question Classification
  [ ] reuters............. The Reuters-21578 benchmark corpus, ApteMod
                           version
Hit Enter to continue:
  [ ] rslp................ RSLP Stemmer (Removedor de Sufixos da Lingua
                           Portuguesa)
  [ ] rte................. PASCAL RTE Challenges 1, 2, and 3
  [ ] sample_grammars..... Sample Grammars
  [ ] semcor.............. SemCor 3.0
  [ ] senseval............ SENSEVAL 2 Corpus: Sense Tagged Text
  [ ] sentence_polarity... Sentence Polarity Dataset v1.0
  [ ] sentiwordnet........ SentiWordNet
  [ ] shakespeare......... Shakespeare XML Corpus Sample
  [ ] sinica_treebank..... Sinica Treebank Corpus Sample
  [ ] smultron............ SMULTRON Corpus Sample
  [ ] snowball_data....... Snowball Data
  [ ] spanish_grammars.... Grammars for Spanish
  [ ] state_union......... C-Span State of the Union Address Corpus
  [*] stopwords........... Stopwords Corpus
  [ ] subjectivity........ Subjectivity Dataset v1.0
  [ ] swadesh............. Swadesh Wordlists
  [ ] switchboard......... Switchboard Corpus Sample
  [ ] tagsets............. Help on Tagsets
  [ ] timit............... TIMIT Corpus Sample
  [ ] toolbox............. Toolbox Sample Files
Hit Enter to continue:
  [ ] treebank............ Penn Treebank Sample
  [ ] twitter_samples..... Twitter Samples
  [ ] udhr2............... Universal Declaration of Human Rights Corpus
                           (Unicode Version)
  [ ] udhr................ Universal Declaration of Human Rights Corpus
  [ ] unicode_samples..... Unicode Samples
  [ ] universal_tagset.... Mappings to the Universal Part-of-Speech Tagset
  [ ] universal_treebanks_v20 Universal Treebanks Version 2.0
  [ ] vader_lexicon....... VADER Sentiment Lexicon
  [ ] verbnet3............ VerbNet Lexicon, Version 3.3
  [ ] verbnet............. VerbNet Lexicon, Version 2.1
  [ ] webtext............. Web Text Corpus
  [ ] wmt15_eval.......... Evaluation data from WMT15
  [ ] word2vec_sample..... Word2Vec Sample
  [ ] wordnet2021......... Open English Wordnet 2021
  [ ] wordnet2022......... Open English Wordnet 2022
  [ ] wordnet31........... Wordnet 3.1
  [ ] wordnet............. WordNet
  [ ] wordnet_ic.......... WordNet-InfoContent
  [ ] words............... Word Lists
  [ ] ycoe................ York-Toronto-Helsinki Parsed Corpus of Old
                           English Prose
Hit Enter to continue:

Collections:
  [P] all-corpora......... All the corpora
  [P] all-nltk............ All packages available on nltk_data gh-pages
                           branch
  [P] all................. All packages
  [P] book................ Everything used in the NLTK Book
  [P] popular............. Popular packages
  [ ] tests............... Packages for running tests
```

```
  [ ] third-party......... Third-party data packages

([*] marks installed packages; [P] marks partially installed collections)

---------------------------------------------------------------------------
    d) Download   l) List    u) Update   c) Config   h) Help   q) Quit
---------------------------------------------------------------------------
Downloader> d

Download which package (l=list; x=cancel)?
  Identifier> stopwords
    Downloading package stopwords to
        C:\Users\17737\AppData\Roaming\nltk_data...
      Package stopwords is already up-to-date!
---------------------------------------------------------------------------
    d) Download   l) List    u) Update   c) Config   h) Help   q) Quit
---------------------------------------------------------------------------
Downloader> q
```

Get the Data

We'll be using a dataset from the UCI datasets! This dataset is already located in the folder for this section.

The file we are using contains a collection of more than 5 thousand SMS phone messages. You can check out the **readme** file for more info.

Let's go ahead and use rstrip() plus a list comprehension to get a list of all the lines of text messages:

In [11]:
```
messages = [line.rstrip() for line in open('smsspamcollection/SMSSpamCollection')]
print(len(messages))
```

```
5574
```

A collection of texts is also sometimes called "corpus". Let's print the first ten messages and number them using **enumerate**:

In [12]:
```
messages[50]
```

Out[12]:
```
'ham\tWhat you thinked about me. First time you saw me in class.'
```

In [13]:
```
for mess_no, message in enumerate(messages[:10]):
    print(mess_no, message)
    print('\n')
```

```
0 ham   Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amor
e wat...


1 ham   Ok lar... Joking wif u oni...


2 spam  Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive entry q
uestion(std txt rate)T&C's apply 08452810075over18's


3 ham   U dun say so early hor... U c already then say...


4 ham   Nah I don't think he goes to usf, he lives around here though


5 spam  FreeMsg Hey there darling it's been 3 week's now and no word back! I'd like some fun you up for it stil
l? Tb ok! XxX std chgs to send, Â£1.50 to rcv


6 ham   Even my brother is not like to speak with me. They treat me like aids patent.


7 ham   As per your request 'Melle Melle (Oru Minnaminunginte Nurungu Vettam)' has been set as your callertune
for all Callers. Press *9 to copy your friends Callertune


8 spam  WINNER!! As a valued network customer you have been selected to receivea Â£900 prize reward! To claim c
all 09061701461. Claim code KL341. Valid 12 hours only.


9 spam  Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for
Free! Call The Mobile Update Co FREE on 08002986030
```

Due to the spacing we can tell that this is a TSV ("tab separated values") file, where the first column is a label saying whether the given message is a normal message (commonly known as "ham") or "spam". The second column is the message itself. (Note our numbers aren't part of the file, they are just from the **enumerate** call).

Using these labeled ham and spam examples, we'll **train a machine learning model to learn to discriminate between ham/spam automatically**. Then, with a trained model, we'll be able to **classify arbitrary unlabeled messages** as ham or spam.

From the official SciKit Learn documentation, we can visualize our process:

Instead of parsing TSV manually using Python, we can just take advantage of pandas! Let's go ahead and import it!

In [14]:
```
import pandas as pd
```

We'll use **read_csv** and make note of the **sep** argument, we can also specify the desired column names by passing in a list of *names*.

In [15]:
```
messages = pd.read_csv('smsspamcollection/SMSSpamCollection', sep='\t',
                           names=["label", "message"])
messages.head()
```

Out[15]:

	label	message
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

Exploratory Data Analysis

Let's check out some of the stats with some plots and the built-in methods in pandas!

In [17]:
```
messages.describe()
```

Out[17]:

	label	message
count	5572	5572
unique	2	5169
top	ham	Sorry, I'll call later
freq	4825	30

Let's use **groupby** to use describe by label, this way we can begin to think about the features that separate ham and spam!

In [19]:
```
messages.groupby('label').describe()
```

Out[19]:

	message			
	count	unique	top	freq
label				
ham	4825	4516	Sorry, I'll call later	30
spam	747	653	Please call our customer service representativ...	4

As we continue our analysis we want to start thinking about the features we are going to be using. This goes along with the general idea of feature engineering. The better your domain knowledge on the data, the better your ability to engineer more features from it. Feature engineering is a very large part of spam detection in general. I encourage you to read up on the topic!

Let's make a new column to detect how long the text messages are:

In [21]:
```
messages['length'] = messages['message'].apply(len)
messages.head()
```

Out[21]:

	label	message	length
0	ham	Go until jurong point, crazy.. Available only ...	111
1	ham	Ok lar... Joking wif u oni...	29
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	155
3	ham	U dun say so early hor... U c already then say...	49
4	ham	Nah I don't think he goes to usf, he lives aro...	61

Data Visualization

Let's visualize this! Let's do the imports:

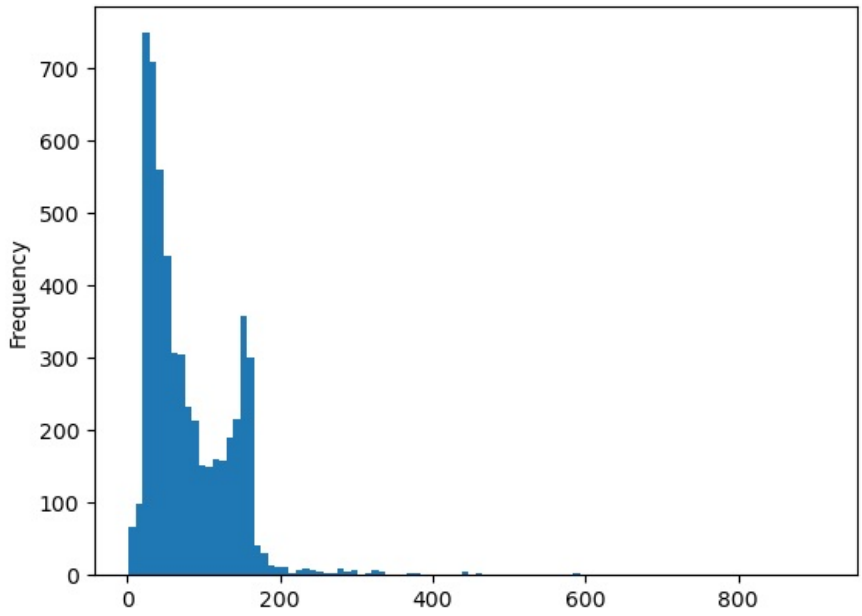
In [22]:
```
import matplotlib.pyplot as plt
```

```
import seaborn as sns

%matplotlib inline
```

In [26]:
```
messages['length'].plot(bins=100, kind='hist')
```

Out[26]:
```
<AxesSubplot:ylabel='Frequency'>
```



Play around with the bin size! Looks like text length may be a good feature to think about! Let's try to explain why the x-axis goes all the way to 1000ish, this must mean that there is some really long message!

In [27]:
```
messages.length.describe()
```

Out[27]:
```
count    5572.000000
mean       80.489950
std        59.942907
min         2.000000
25%        36.000000
50%        62.000000
75%       122.000000
max       910.000000
Name: length, dtype: float64
```

Woah! 910 characters, let's use masking to find this message:

In [28]:
```
messages[messages['length'] == 910]
```

Out[28]:

	label	message	length
1085	ham	For me the love should start with attraction.i...	910

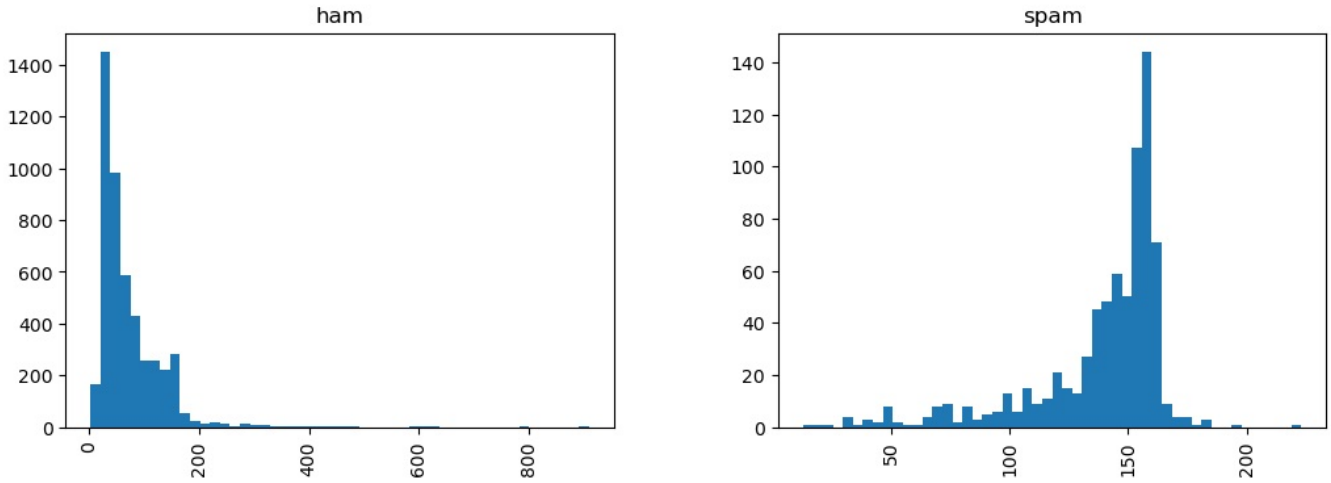
In [29]:
```
messages[messages['length'] == 910]['message'].iloc[0]
```

Out[29]:
```
"For me the love should start with attraction.i should feel that I need her every time around me.she should be
the first thing which comes in my thoughts.I would start the day and end it with her.she should be there every
time I dream.love will be then when my every breath has her name.my life should happen around her.my life will
be named to her.I would cry for her.will give all my happiness and take all her sorrows.I will be ready to figh
t with anyone for her.I will be in love when I will be doing the craziest things for her.love will be when I do
n't have to proove anyone that my girl is the most beautiful lady on the whole planet.I will always be singing
praises for her.love will be when I start up making chicken curry and end up makiing sambar.life will be the mo
st beautiful then.will get every morning and thank god for the day because she is with me.I would like to say a
lot..will tell later.."
```

Looks like we have some sort of Romeo sending texts! But let's focus back on the idea of trying to see if message length is a distinguishing feature between ham and spam:

In [30]:
```
messages.hist(column='length', by='label', bins=50,figsize=(12,4))
```

Out[30]:
```
array([<AxesSubplot:title={'center':'ham'}>,
       <AxesSubplot:title={'center':'spam'}>], dtype=object)
```



Very interesting! Through just basic EDA we've been able to discover a trend that spam messages tend to have more characters. (Sorry Romeo!)

Now let's begin to process the data so we can eventually use it with SciKit Learn!

Text Pre-processing

Our main issue with our data is that it is all in text format (strings). The classification algorithms that we've learned about so far will need some sort of numerical feature vector in order to perform the classification task. There are actually many methods to convert a corpus to a vector format. The simplest is the the bag-of-words approach, where each unique word in a text will be represented by one number.

In this section we'll convert the raw messages (sequence of characters) into vectors (sequences of numbers).

As a first step, let's write a function that will split a message into its individual words and return a list. We'll also remove very common words, ('the', 'a', etc..). To do this we will take advantage of the NLTK library. It's pretty much the standard library in Python for processing text and has a lot of useful features. We'll only use some of the basic ones here.

Let's create a function that will process the string in the message column, then we can just use **apply()** in pandas do process all the text in the DataFrame.

First removing punctuation. We can just take advantage of Python's built-in **string** library to get a quick list of all the possible punctuation:

In [33]:
```
import string

mess = 'Sample message! Notice: it has punctuation.'

# Check characters to see if they are in punctuation
nopunc = [char for char in mess if char not in string.punctuation]
```

In [34]:
```
string.punctuation
```

Out[34]:
```
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

In [35]:
```
nopunc
```

Out[35]:
```
['S',
 'a',
 'm',
 'p',
 'l',
 'e',
 ' ',
 'm',
 'e',
 's',
 's',
 'a',
 'g',
 'e',
 ' ',
 'N',
 'o',
 't',
 'i',
 'c',
 'e',
 ' ',
 'i',
 't',
 ' ',
 'h',
 'a',
 's',
 ' ',
 'p',
 'u',
 'n',
 'c',
 't',
 'u',
 'a',
 't',
 'i',
 'o',
 'n']
```

In [36]:
```
# Join the characters again to form the string.
nopunc = ''.join(nopunc)
```

In [39]:
```
nopunc
```

Out[39]:
```
'Sample message Notice it has punctuation'
```

Now let's see how to remove stopwords. We can impot a list of english stopwords from NLTK (check the documentation for more languages and info).

In [37]:
```
from nltk.corpus import stopwords
stopwords.words('english')
```

Out[37]:
```
['i',
 'me',
 'my',
 'myself',
 'we',
 'our',
 'ours',
 'ourselves',
 'you',
 "you're",
 "you've",
 "you'll",
 "you'd",
 'your',
 'yours',
 'yourself',
 'yourselves',
 'he',
 'him',
 'his',
 'himself',
 'she',
 "she's",
 'her',
 'hers',
 'herself',
 'it',
 "it's",
 'its',
 'itself',
 'they',
 'them',
 'their',
```

```
'theirs',
'themselves',
'what',
'which',
'who',
'whom',
'this',
'that',
"that'll",
'these',
'those',
'am',
'is',
'are',
'was',
'were',
'be',
'been',
'being',
'have',
'has',
'had',
'having',
'do',
'does',
'did',
'doing',
'a',
'an',
'the',
'and',
'but',
'if',
'or',
'because',
'as',
'until',
'while',
'of',
'at',
'by',
'for',
'with',
'about',
'against',
'between',
'into',
'through',
'during',
'before',
'after',
'above',
'below',
'to',
'from',
'up',
'down',
'in',
'out',
'on',
'off',
'over',
'under',
'again',
'further',
'then',
'once',
'here',
'there',
'when',
'where',
'why',
'how',
'all',
'any',
'both',
'each',
'few',
'more',
'most',
'other',
'some',
'such',
'no',
'nor',
'not',
'only',
'own',
'same',
```

```
 'so',
 'than',
 'too',
 'very',
 's',
 't',
 'can',
 'will',
 'just',
 'don',
 "don't",
 'should',
 "should've",
 'now',
 'd',
 'll',
 'm',
 'o',
 're',
 've',
 'y',
 'ain',
 'aren',
 "aren't",
 'couldn',
 "couldn't",
 'didn',
 "didn't",
 'doesn',
 "doesn't",
 'hadn',
 "hadn't",
 'hasn',
 "hasn't",
 'haven',
 "haven't",
 'isn',
 "isn't",
 'ma',
 'mightn',
 "mightn't",
 'mustn',
 "mustn't",
 'needn',
 "needn't",
 'shan',
 "shan't",
 'shouldn',
 "shouldn't",
 'wasn',
 "wasn't",
 'weren',
 "weren't",
 'won',
 "won't",
 'wouldn',
 "wouldn't"]
```

In [42]:
```
stopwords.words('english')[0:10] # Show some stop word
```

Out[42]:
```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're"]
```

In [41]:
```
nopunc.split()
```

Out[41]:
```
['Sample', 'message', 'Notice', 'it', 'has', 'punctuation']
```

In [44]:
```
# Now just remove any stopwords
clean_mess = [word for word in nopunc.split() if word.lower() not in stopwords.words('english')]
```

In [45]:
```
clean_mess
```

Out[45]:
```
['Sample', 'message', 'Notice', 'punctuation']
```

Now let's put both of these together in a function to apply it to our DataFrame later on:

In [46]:
```
def text_process(mess):
    """
    Takes in a string of text, then performs the following:
    1. Remove all punctuation
    2. Remove all stopwords
    3. Returns a list of the cleaned text
    """
    # Check characters to see if they are in punctuation
    nopunc = [char for char in mess if char not in string.punctuation]

    # Join the characters again to form the string.
```

```
    nopunc = ''.join(nopunc)

    # Now just remove any stopwords
    return [word for word in nopunc.split() if word.lower() not in stopwords.words('english')]
```

Here is the original DataFrame again:

In [47]:
```
messages.head()
```

Out[47]:

	label	message	length
0	ham	Go until jurong point, crazy.. Available only ...	111
1	ham	Ok lar... Joking wif u oni...	29
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	155
3	ham	U dun say so early hor... U c already then say...	49
4	ham	Nah I don't think he goes to usf, he lives aro...	61

Now let's "tokenize" these messages. Tokenization is just the term used to describe the process of converting the normal text strings in to a list of tokens (words that we actually want).

Let's see an example output on on column:

Note: We may get some warnings or errors for symbols we didn't account for or that weren't in Unicode (like a British pound symbol)

In [48]:
```
# Check to make sure its working
messages['message'].head(5).apply(text_process)
```

Out[48]:
```
0    [Go, jurong, point, crazy, Available, bugis, n...
1                        [Ok, lar, Joking, wif, u, oni]
2    [Free, entry, 2, wkly, comp, win, FA, Cup, fin...
3        [U, dun, say, early, hor, U, c, already, say]
4    [Nah, dont, think, goes, usf, lives, around, t...
Name: message, dtype: object
```

In [49]:
```
# Show original dataframe
messages.head()
```

Out[49]:

	label	message	length
0	ham	Go until jurong point, crazy.. Available only ...	111
1	ham	Ok lar... Joking wif u oni...	29
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	155
3	ham	U dun say so early hor... U c already then say...	49
4	ham	Nah I don't think he goes to usf, he lives aro...	61

Continuing Normalization

There are a lot of ways to continue normalizing this text. Such as Stemming or distinguishing by part of speech.

NLTK has lots of built-in tools and great documentation on a lot of these methods. Sometimes they don't work well for text-messages due to the way a lot of people tend to use abbreviations or shorthand, For example:

```
'Nah dawg, IDK! Wut time u headin to da club?'
```

versus

```
'No dog, I don't know! What time are you heading to the club?'
```

Some text normalization methods will have trouble with this type of shorthand and so I'll leave you to explore those more advanced methods through the NLTK book online.

For now we will just focus on using what we have to convert our list of words to an actual vector that SciKit-Learn can use.

Vectorization

Currently, we have the messages as lists of tokens (also known as lemmas) and now we need to convert each of those messages into a vector the SciKit Learn's algorithm models can work with.

Now we'll convert each message, represented as a list of tokens (lemmas) above, into a vector that machine learning models can understand.

We'll do that in three steps using the bag-of-words model:

1. Count how many times does a word occur in each message (Known as term frequency)
2. Weigh the counts, so that frequent tokens get lower weight (inverse document frequency)
3. Normalize the vectors to unit length, to abstract from the original text length (L2 norm)

Let's begin the first step:

Each vector will have as many dimensions as there are unique words in the SMS corpus. We will first use SciKit Learn's **CountVectorizer**. This model will convert a collection of text documents to a matrix of token counts.

We can imagine this as a 2-Dimensional matrix. Where the 1-dimension is the entire vocabulary (1 row per word) and the other dimension are the actual documents, in this case a column per text message.

For example:

	Message 1	Message 2	...	Message N
Word 1 Count	0	1	...	0
Word 2 Count	0	0	...	0
...	1	2	...	0
Word N Count	0	1	...	1

Since there are so many messages, we can expect a lot of zero counts for the presence of that word in that document. Because of this, SciKit Learn will output a Sparse Matrix.

In [50]:
```
from sklearn.feature_extraction.text import CountVectorizer
```

There are a lot of arguments and parameters that can be passed to the CountVectorizer. In this case we will just specify the **analyzer** to be our own previously defined function:

In [51]:
```
# Might take awhile...
bow_transformer = CountVectorizer(analyzer=text_process).fit(messages['message'])

# Print total number of vocab words
print(len(bow_transformer.vocabulary_))
```

```
11425
```

Let's take one text message and get its bag-of-words counts as a vector, putting to use our new `bow_transformer`:

In [52]:
```
message4 = messages['message'][3]
print(message4)
```

```
U dun say so early hor... U c already then say...
```

Now let's see its vector representation:

In [53]:
```
bow4 = bow_transformer.transform([message4])
print(bow4)
print(bow4.shape)
```

```
  (0, 4068)     2
  (0, 4629)     1
  (0, 5261)     1
  (0, 6204)     1
  (0, 6222)     1
  (0, 7186)     1
  (0, 9554)     2
(1, 11425)
```

This means that there are seven unique words in message number 4 (after removing common stop words). Two of them appear twice, the rest only once. Let's go ahead and check and confirm which ones appear twice:

In [55]:
```
print(bow_transformer.get_feature_names()[4068])
print(bow_transformer.get_feature_names()[9554])
```

```
U
say
```

Now we can use **.transform** on our Bag-of-Words (bow) transformed object and transform the entire DataFrame of messages. Let's go ahead and check out how the bag-of-words counts for the entire SMS corpus is a large, sparse matrix:

In [56]:
```
messages_bow = bow_transformer.transform(messages['message'])
```

In [57]:
```
print('Shape of Sparse Matrix: ', messages_bow.shape)
print('Amount of Non-Zero occurences: ', messages_bow.nnz)
```

```
Shape of Sparse Matrix:  (5572, 11425)
Amount of Non-Zero occurences:  50548
```

In [58]:
```
#More complex
sparsity = (100.0 * messages_bow.nnz / (messages_bow.shape[0] * messages_bow.shape[1]))
print('sparsity: {}'.format(round(sparsity)))
```

```
sparsity: 0
```

After the counting, the term weighting and normalization can be done with TF-IDF, using scikit-learn's `TfidfTransformer`.

So what is TF-IDF?

TF-IDF stands for *term frequency-inverse document frequency*, and the tf-idf weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the tf-idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query.

One of the simplest ranking functions is computed by summing the tf-idf for each query term; many more sophisticated ranking functions are variants of this simple model.

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

TF: Term Frequency, which measures how frequently a term occurs in a document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. Thus, the term frequency is often divided by the document length (aka. the total number of terms in the document) as a way of normalization:

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

IDF: Inverse Document Frequency, which measures how important a term is. While computing TF, all terms are considered equally important. However it is known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus we need to weigh down the frequent terms while scale up the rare ones, by computing the following:

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

See below for a simple example.

Example:

Consider a document containing 100 words wherein the word cat appears 3 times.

The term frequency (i.e., tf) for cat is then (3 / 100) = 0.03. Now, assume we have 10 million documents and the word cat appears in one thousand of these. Then, the inverse document frequency (i.e., idf) is calculated as log(10,000,000 / 1,000) = 4. Thus, the Tf-idf weight is the product of these quantities: 0.03 * 4 = 0.12.

Let's go ahead and see how we can do this in SciKit Learn:

In [59]:
```
from sklearn.feature_extraction.text import TfidfTransformer

tfidf_transformer = TfidfTransformer().fit(messages_bow)
tfidf4 = tfidf_transformer.transform(bow4)
print(tfidf4)
```

```
  (0, 9554)     0.5385626262927564
  (0, 7186)     0.4389365653379857
  (0, 6222)     0.3187216892949149
  (0, 6204)     0.29953799723697416
  (0, 5261)     0.29729957405868723
  (0, 4629)     0.26619801906087187
  (0, 4068)     0.40832589933384067
```

We'll go ahead and check what is the IDF (inverse document frequency) of the word `"u"` and of word `"university"`?

In [62]:
```
print(tfidf_transformer.idf_[bow_transformer.vocabulary_['u']])
print(tfidf_transformer.idf_[bow_transformer.vocabulary_['university']])
```

```
3.2800524267409408
8.527076498901426
```

To transform the entire bag-of-words corpus into TF-IDF corpus at once:

In [63]:
```
messages_tfidf = tfidf_transformer.transform(messages_bow)
print(messages_tfidf.shape)
```

```
(5572, 11425)
```

There are many ways the data can be preprocessed and vectorized. These steps involve feature engineering and building a "pipeline". I encourage you to check out SciKit Learn's documentation on dealing with text data as well as the expansive collection of available papers and books on the general topic of NLP.

Training a model

With messages represented as vectors, we can finally train our spam/ham classifier. Now we can actually use almost any sort of classification algorithms. For a variety of reasons, the Naive Bayes classifier algorithm is a good choice.

We'll be using scikit-learn here, choosing the Naive Bayes classifier to start with:

In [64]:
```
from sklearn.naive_bayes import MultinomialNB
spam_detect_model = MultinomialNB().fit(messages_tfidf, messages['label'])
```

Let's try classifying our single random message and checking how we do:

In [65]:
```
print('predicted:', spam_detect_model.predict(tfidf4)[0])
print('expected:', messages.label[3])
```

```
predicted: ham
expected: ham
```

Fantastic! We've developed a model that can attempt to predict spam vs ham classification!

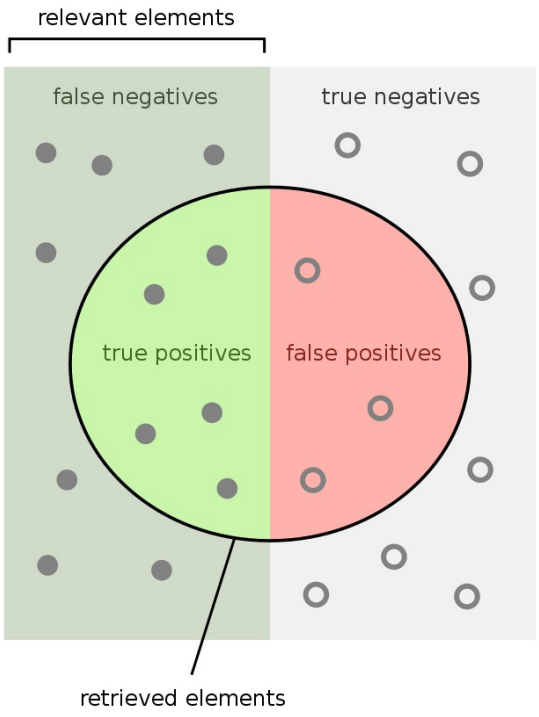
Part 6: Model Evaluation

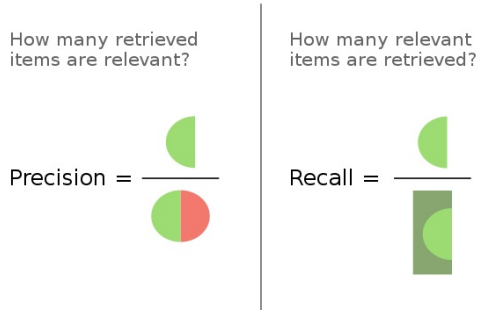
Now we want to determine how well our model will do overall on the entire dataset. Let's begin by getting all the predictions:

In [66]:
```
all_predictions = spam_detect_model.predict(messages_tfidf)
print(all_predictions)
```

```
['ham' 'ham' 'spam' ... 'ham' 'ham' 'ham']
```

We can use SciKit Learn's built-in classification report, which returns precision, recall, f1-score, and a column for support (meaning how many cases supported that classification). Check out the links for more detailed info on each of these metrics and the figure below:





In [67]:
```
from sklearn.metrics import classification_report
print (classification_report(messages['label'], all_predictions))
```

```
              precision    recall  f1-score   support

         ham       0.98      1.00      0.99      4825
        spam       1.00      0.85      0.92       747

    accuracy                           0.98      5572
   macro avg       0.99      0.92      0.95      5572
weighted avg       0.98      0.98      0.98      5572
```

There are quite a few possible metrics for evaluating model performance. Which one is the most important depends on the task and the business effects of decisions based off of the model. For example, the cost of mis-predicting "spam" as "ham" is probably much lower than mis-predicting "ham" as "spam".

In the above "evaluation",we evaluated accuracy on the same data we used for training. **You should never actually evaluate on the same dataset you train on!**

Such evaluation tells us nothing about the true predictive power of our model. If we simply remembered each example during training, the accuracy on training data would trivially be 100%, even though we wouldn't be able to classify any new messages.

A proper way is to split the data into a training/test set, where the model only ever sees the **training data** during its model fitting and parameter tuning. The **test data** is never used in any way. This is then our final evaluation on test data is representative of true predictive performance.

Train Test Split

In [68]:
```
from sklearn.model_selection import train_test_split

msg_train, msg_test, label_train, label_test = \
train_test_split(messages['message'], messages['label'], test_size=0.2)

print(len(msg_train), len(msg_test), len(msg_train) + len(msg_test))
```

```
4457 1115 5572
```

The test size is 20% of the entire dataset (1115 messages out of total 5572), and the training is the rest (4457 out of 5572). Note the default split would have been 30/70.

Creating a Data Pipeline

Let's run our model again and then predict off the test set. We will use SciKit Learn's pipeline capabilities to store a pipeline of workflow. This will allow us to set up all the transformations that we will do to the data for future use. Let's see an example of how it works:

In [69]:
```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ('bow', CountVectorizer(analyzer=text_process)),  # strings to token integer counts
    ('tfidf', TfidfTransformer()),  # integer counts to weighted TF-IDF scores
    ('classifier', MultinomialNB()),  # train on TF-IDF vectors w/ Naive Bayes classifier
])
```

Now we can directly pass message text data and the pipeline will do our pre-processing for us! We can treat it as a model/estimator API:

In [70]:
```
pipeline.fit(msg_train,label_train)
```

Out[70]:
```
Pipeline(steps=[('bow',
                 CountVectorizer(analyzer=<function text_process at 0x0000021B5390DCA0>)),
                ('tfidf', TfidfTransformer()),
                ('classifier', MultinomialNB())])
```

In [71]:
```
predictions = pipeline.predict(msg_test)
```

In [72]:
```
print(classification_report(predictions,label_test))
```

```
              precision    recall  f1-score   support

         ham       1.00      0.95      0.97      1011
        spam       0.67      1.00      0.80       104

    accuracy                           0.95      1115
   macro avg       0.84      0.97      0.89      1115
weighted avg       0.97      0.95      0.96      1115
```

Now we have a classification report for our model on a true testing set! There is a lot more to Natural Language Processing than what

Now we have a classification report for our model on a true testing set! There is a lot more to Natural Language Processing than what we've covered here, and its vast expanse of topic could fill up several college courses! I encourage you to check out the resources below for more information on NLP!

More Resources

Check out the links below for more info on Natural Language Processing:

NLTK Book Online

Kaggle Walkthrough

SciKit Learn's Tutorial

Good Job!