


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 [1]:
```
# ONLY RUN THIS CELL IF YOU NEED
# TO DOWNLOAD NLTK AND HAVE CONDA
# WATCH THE VIDEO FOR FULL INSTRUCTIONS ON THIS STEP

# Uncomment the code below and run:


# !conda install nltk #This installs nltk
# import nltk # Imports the library
# nltk.download() #Download the necessary datasets
```

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 [3]:
```
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 [4]:
```
for message_no, message in enumerate(messages[:10]):
    print(message_no, message)
    print('\n')
```

```
0 ham   Go until jurong point, crazy.. Available only in bugis n great world la e
buffet... Cine there got amore 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 question(std txt rate)T&C's apply 08452810075over1
8'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 li
ke some fun you up for it still? Tb ok! XxX std chgs to send, £1.50 to rcv


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


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


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


9 spam  Had your mobile 11 months or more? U R entitled to Update to the latest c
olour 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:

 No description has been provided for this image

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

In [6]:
```
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 [7]:
```
messages = pd.read_csv('smsspamcollection/SMSSpamCollection', sep='\t',
                           names=["label", "message"])
messages.head()
```

Out[7]:

	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 [8]:
```
messages.describe()
```

Out[8]:

	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 [9]:
```
messages.groupby('label').describe()
```

Out[9]:

		message
label		
ham	**count**	4825
	unique	4516
	top	Sorry, I'll call later
	freq	30
spam	**count**	747
	unique	653
	top	Please call our customer service representativ...
	freq	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 [10]:
```
messages['length'] = messages['message'].apply(len)
messages.head()
```

Out[10]:

	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

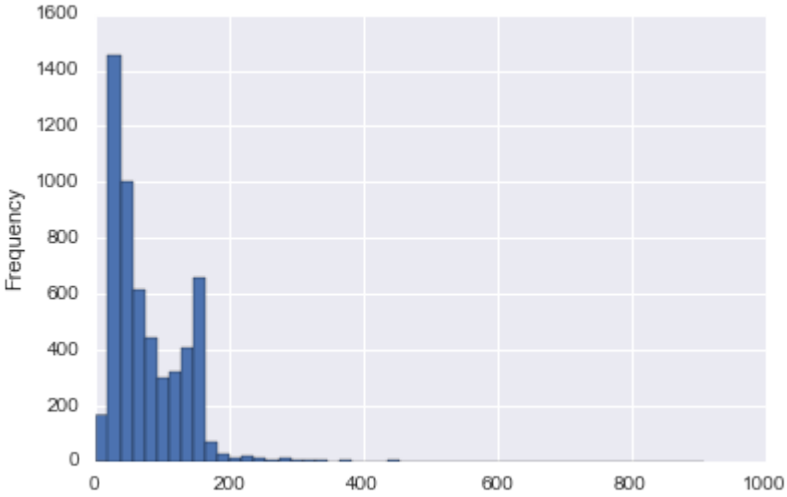
Let's visualize this! Let's do the imports:

In [11]:
```
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

In [12]:
```
messages['length'].plot(bins=50, kind='hist')
```

Out[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11a03c0b8>
```



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 [13]:
```
messages.length.describe()
```

Out[13]:
```
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:

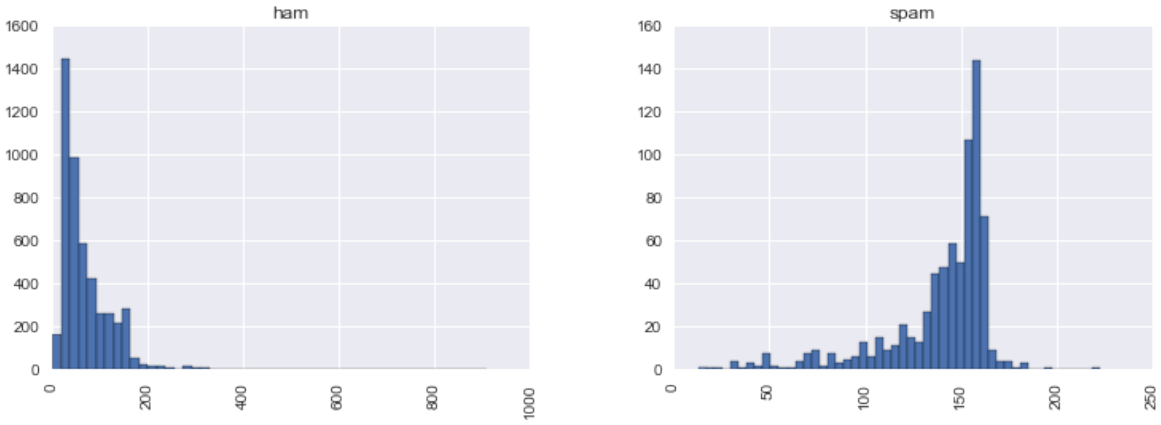
In [14]:
```
messages[messages['length'] == 910]['message'].iloc[0]
```

Out[14]:
```
"For me the love should start with attraction.i should feel that I need her eve
ry time around me.she should be the first thing which comes in my thoughts.I wo
uld start the day and end it with her.she should be there every time I dream.lo
ve 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 fight 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 ma
king chicken curry and end up makiing sambar.life will be the most beautiful th
en.will get every morning and thank god for the day because she is with me.I wo
uld 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 [18]:
```
messages.hist(column='length', by='label', bins=50,figsize=(12,4))
```

Out[18]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x11d089cc0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11d339908>], dtype=ob
ject)
```



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 [19]:
```
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]

# Join the characters again to form the string.
nopunc = ''.join(nopunc)
```

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 [20]:
```
from nltk.corpus import stopwords
stopwords.words('english')[0:10] # Show some stop words
```

Out[20]:
```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your']
```

In [21]:
```
nopunc.split()
```

Out[21]:
```
['Sample', 'message', 'Notice', 'it', 'has', 'punctuation']
```

In [22]:
```
# Now just remove any stopwords
clean_mess = [word for word in nopunc.split() if word.lower() not in stopwords.w
```

In [23]:
```
clean_mess
```

Out[23]:
```
['Sample', 'message', 'Notice', 'punctuation']
```

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

In [24]:
```
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.wor
```

Here is the original DataFrame again:

In [25]:
```
messages.head()
```

Out[25]:

	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 [26]:
```
# Check to make sure its working
messages['message'].head(5).apply(text_process)
```

Out[26]:
```
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 [27]:
```
# Show original dataframe
messages.head()
```

Out[27]:

	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 [28]:
```
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 [31]:
```
# Might take awhile...
bow_transformer = CountVectorizer(analyzer=text_process).fit(messages['message']
```

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

```
11444
```

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

In [32]:
```
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 [34]:
```
bow4 = bow_transformer.transform([message4])
print(bow4)
print(bow4.shape)
```

```
  (0, 4073)     2
  (0, 4638)     1
  (0, 5270)     1
  (0, 6214)     1
  (0, 6232)     1
  (0, 7197)     1
  (0, 9570)     2
(1, 11444)
```

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 [36]:
```
print(bow_transformer.get_feature_names()[4073])
print(bow_transformer.get_feature_names()[9570])
```

```
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 [39]:
```
messages_bow = bow_transformer.transform(messages['message'])
```

In [40]:
```
print('Shape of Sparse Matrix: ', messages_bow.shape)
print('Amount of Non-Zero occurences: ', messages_bow.nnz)
```

```
Shape of Sparse Matrix:  (5572, 11444)
Amount of Non-Zero occurences:  50795
```

In [46]:
```
sparsity = (100.0 * messages_bow.nnz / (messages_bow.shape[0] * messages_bow.sha
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 [48]:
```
from sklearn.feature_extraction.text import TfidfTransformer

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

```
  (0, 9570)     0.538562626293
  (0, 7197)     0.438936565338
  (0, 6232)     0.318721689295
  (0, 6214)     0.299537997237
  (0, 5270)     0.297299574059
  (0, 4638)     0.266198019061
  (0, 4073)     0.408325899334
```

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

In [50]:
```
print(tfidf_transformer.idf_[bow_transformer.vocabulary_['u']])
print(tfidf_transformer.idf_[bow_transformer.vocabulary_['university']])
```

```
3.28005242674
8.5270764989
```

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

In [51]:
```
messages_tfidf = tfidf_transformer.transform(messages_bow)
print(messages_tfidf.shape)
```

```
(5572, 11444)
```

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 [52]:
```
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 [54]:
```
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!

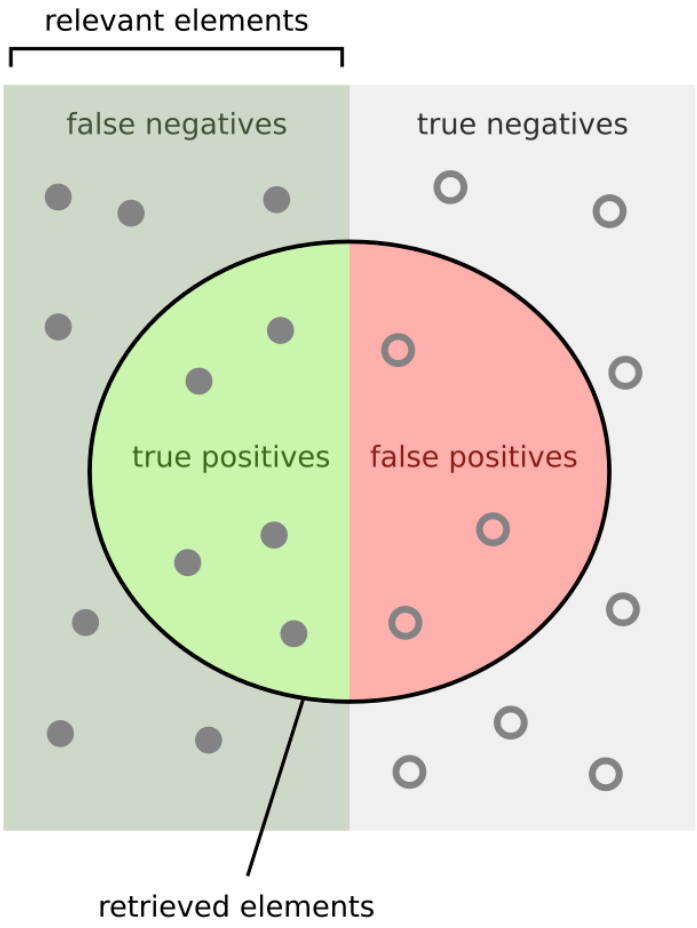
Part 6: Model Evaluation

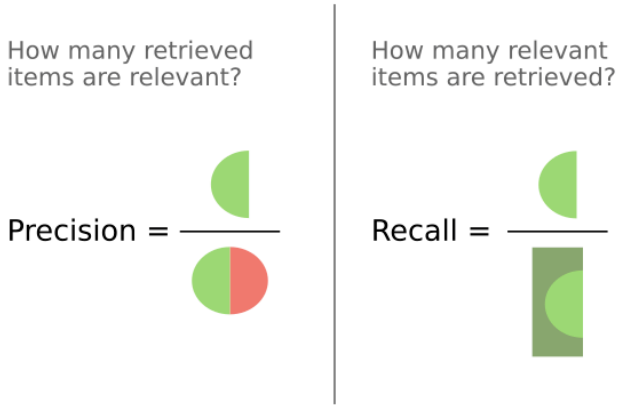
Now we want to determine how well our model will do overall on the entire dataset. Let's begin by getting all the predictions:

In [55]:
```
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 [56]:
```
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

avg / total       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 [57]:
```
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 [58]:
```
from sklearn.pipeline import Pipeline

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

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 [59]:
```
pipeline.fit(msg_train,label_train)
```

Out[59]:
```
Pipeline(steps=[('bow', CountVectorizer(analyzer=<function text_process at 0x11
e795bf8>, binary=False,
        decode_error='strict', dtype=<class 'numpy.int64'>,
        encoding='utf-8', input='content', lowercase=True, max_df=1.0,
        max_features=None, min_df=1, ngram_range=(1, 1), preprocessor=None,...f
=False, use_idf=True)), ('classifier', MultinomialNB(alpha=1.0, class_prior=Non
e, fit_prior=True))])
```

In [60]:
```
predictions = pipeline.predict(msg_test)
```

In [61]:
```
print(classification_report(predictions,label_test))
```

```
             precision    recall  f1-score   support

        ham       1.00      0.96      0.98      1001
       spam       0.75      1.00      0.85       114

avg / total       0.97      0.97      0.97      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 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!