Lab 5 Sparse Spam Classification

This notebook provides an example of an app to classify phone SMS messages as either "spam" or "ham" (=not spam). Some of this content has been adapted from a tutorial by Radimre Hurek: https://radimrehurek.com/data_science_python/ (https://radimrehurek.com/data_science_python/) and has been updated by Dr. Riley.

Please follow through this notebook linearly and insert your modifications and additions appropriately throughout. You will also need to update some of the existing cells to conform to the style expectations of the checklist.

Introductory Notes:

As mentioned in the above cell the majority of this notebook has been created by Dr. Riley, however due to the fact that this notebook was to serve as a tutorial to create a spam classifier using a sparse matrix, this notebook has been further modified by Nigel Nelson in exploration of this spam classifier.

Modified by: Nigel Nelson

Lets start with importing some things...

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import csv
from textblob import TextBlob
import pandas as pd
import sklearn
import nltk
import re
import sys
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.metrics import precision_score
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /home/ad.msoe.edu/nelsonni/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /home/ad.msoe.edu/nelsonni/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package wordnet to
[nltk_data]     /home/ad.msoe.edu/nelsonni/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

Part 1: Load data, explore

Let's get the dataset and put it in the data folder.

This file contains **a collection of more than 5 thousand SMS phone messages** (see the `readme` file for more info). First, load them using Pandas with one column named `label` and one named `message` ...

In [2]:
```
messages = pd.read_csv('/data/cs2300/L5/SMSSpamCollection.txt', sep='\t', quot
ing=csv.QUOTE_NONE,
                           names=["label", "message"])
messages.head()
```

Out[2]:

	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...

You should take a look at the basic statistics for this dataset using Pandas describe() method

Following cell modified by Nigel Nelson:

In [3]:
```
messages.groupby('label').describe()
```

Out[3]:

	message			
	count	unique	top	freq
label				
ham	4827	4518	Sorry, I'll call later	30
spam	747	653	Please call our customer service representativ...	4

add a Pandas column that describes the length of the messages

Following cell modified by Nigel Nelson:

In [4]:
```
messages['length'] = messages['message'].apply(lambda s: len(s))
messages.head()
```

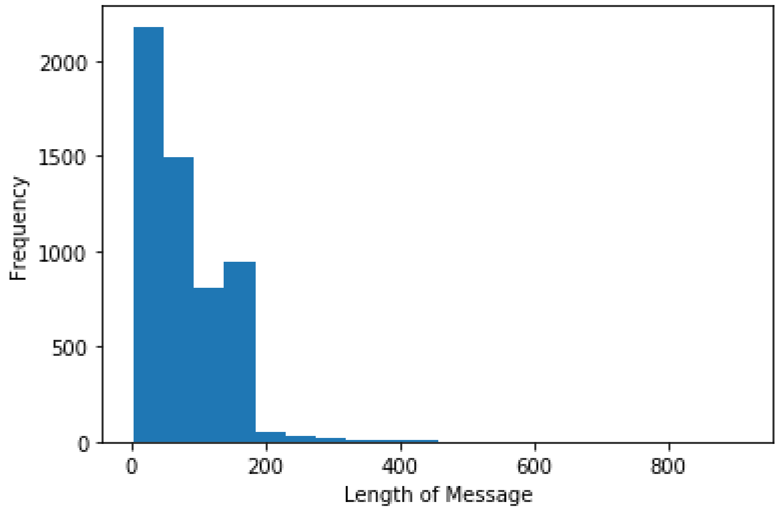
Out[4]:

	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

This will allow you to run the cell below to make a histogram of the length.

In [5]:
```
ax = messages.length.plot(bins=20, kind='hist')
ax.set_xlabel('Length of Message')
```

Out[5]:
```
Text(0.5, 0, 'Length of Message')
```



In [6]:
```
messages.length.describe()
```

Out[6]:
```
count    5574.000000
mean       80.478292
std        59.848302
min         2.000000
25%        36.000000
50%        62.000000
75%       122.000000
max       910.000000
Name: length, dtype: float64
```

Find and print that really long one...

Following cell modified by Nigel Nelson:

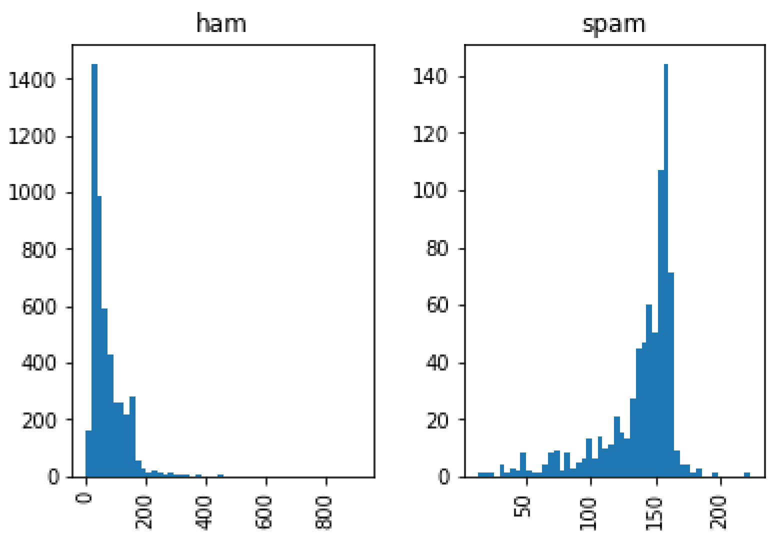
In [7]:
```
print(messages.loc[messages['length'] == 910].message)
```

```
1085    For me the love should start with attraction.i...
Name: message, dtype: object
```

We can see if there is there any difference in message length between spam and ham by running the following code to plot them side by side.

In [8]:
```
messages.hist(column='length', by='label', bins=50)
```

Out[8]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f86b7c75850>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f86b73398d0>],
      dtype=object)
```



Great, but this is not sufficient for us to create a classifier. We need machine learning!

Part 2: Data preprocessing

Next we convert the raw messages (sequence of characters) into vectors (sequences of numbers).

The mapping is not 1-to-1; we'll use the bag-of-words (http://en.wikipedia.org/wiki/Bag-of-words_model) approach, where each unique word in a text will be represented by one number.

As a first step, here is a function that will split a message into its individual words:

In [9]:
```
def split_into_tokens(message):
    return TextBlob(message).words
```

You should tokenize them by applying the split_into_tokens method to the message column of the dataframe in the following cell. Print the results to convince yourself that they are correct. You do not need to store these results back in the dataframe.

Following cell modified by Nigel Nelson:

In [10]:
```
messages.message.apply(lambda s: split_into_tokens(s)).head()
```

Out[10]:
```
0    [Go, until, jurong, point, crazy, Available, o...
1                        [Ok, lar, Joking, wif, u, oni]
2    [Free, entry, in, 2, a, wkly, comp, to, win, F...
3    [U, dun, say, so, early, hor, U, c, already, t...
4    [Nah, I, do, n't, think, he, goes, to, usf, he...
Name: message, dtype: object
```

With textblob, we can detect part-of-speech (POS) (http://www.ling.upenn.edu/courses/Fall_2007/ling001/penn_treebank_pos.html) tags with:

In [11]:
```
TextBlob("Hello world, how is it going?").tags  # list of (word, POS) pairs
```

Out[11]:
```
[('Hello', 'NNP'),
 ('world', 'NN'),
 ('how', 'WRB'),
 ('is', 'VBZ'),
 ('it', 'PRP'),
 ('going', 'VBG')]
```

In [12]:
```
def split_into_lemmas(message):
    words = TextBlob(message).words
    # for each word, take its "base form" = lemma
    return [word.lemma for word in words]
```

Normalize words into their base form (lemmas (http://en.wikipedia.org/wiki/Lemmatisation)) by applying the split_into_lemmas function below to the message column of the dataframe. Again, you do not need to store these results, so you can use `.head()` to view the output.

Following cell modified by Nigel Nelson:

In [13]:
```
messages.message.apply(lambda s: split_into_lemmas(s)).head()
```

Out[13]:
```
0    [Go, until, jurong, point, crazy, Available, o...
1                        [Ok, lar, Joking, wif, u, oni]
2    [Free, entry, in, 2, a, wkly, comp, to, win, F...
3    [U, dun, say, so, early, hor, U, c, already, t...
4    [Nah, I, do, n't, think, he, go, to, usf, he, ...
Name: message, dtype: object
```

You can probably think of many more ways to improve the preprocessing: decoding HTML entities (those `&amp;` and `&lt;` we saw above); filtering out stop words (pronouns etc); adding more features, such as an word-in-all-caps indicator and so on. So keep those in mind for later...

Part 3: Data to vectors

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

Doing that requires essentially three steps, in the bag-of-words model:

1. counting how many times does a word occur in each message (term frequency)
2. weighting the counts, so that frequent tokens get lower weight (inverse document frequency)
3. normalizing the vectors to unit length, to abstract from the original text length

Each vector has as many dimensions as there are unique words in the SMS corpus. We can count the number of unique words using the following cell...

In [14]:
```
bow_transformer = CountVectorizer(analyzer=split_into_lemmas).fit(messages['me
ssage'])
print(len(bow_transformer.vocabulary_))
```

```
11010
```

Here we used `scikit-learn` (`sklearn`), a powerful Python library for teaching machine learning. It contains a multitude of various methods and options.

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

In [15]:
```
message4 = messages['message'][3]
print(message4)
```

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

In [16]:
```
bow4 = bow_transformer.transform([message4])
print(bow4)
print(bow4.shape)
```

```
  (0, 4189)	2
  (0, 4762)	1
  (0, 5363)	1
  (0, 6219)	1
  (0, 6243)	1
  (0, 7137)	1
  (0, 9280)	2
  (0, 9589)	1
  (0, 10054)	1
(1, 11010)
```

So, nine unique words are in this message. Two of them appear twice, the rest only once.

Write some code in the next cell that identifies the words that appear twice. You are encouraged to use the CountVectorizer's get_feature_names() method to make this easier

*Following **2** cells modified by Nigel Nelson:*

```
In [17]: bow_transformer.get_feature_names()[4189]
```

```
Out[17]: 'U'
```

```
In [18]: bow_transformer.get_feature_names()[9280]
```

```
Out[18]: 'say'
```

The bag-of-words counts for the entire SMS corpus are a large, sparse matrix (generated using `bow_transformer.transform()` on the appropriate dataframe column). In the following cell, calculate the sparsity using `.nnz` and the shape.

*Following **4** cells modified by Nigel Nelson:*

```
In [19]: messages_bow = bow_transformer.transform(messages.message)
```

Dimensions of Sparse Matrix:

The below cell returns a tuple of the dimensions of the created sparse matrix, where the first value is the number rows of the matrix which are composed of the individual messages, and the second value is the number of columns which is the individual tokens found in the messages

```
In [20]: messages_bow.shape
```

```
Out[20]: (5574, 11010)
```

Calculation of Sparsity:

The below cell calcualtes the sparsity of the created sparse matrix, or the percentage of the matrix that contains the value 0. This is done by dividing the number of non-zero entries by the number of elements in the sparse matrix, accomplished by multiplying the number of rows by the number of columns. That number is then subtracted from 1.0 to give the percentage of the matrix that contains the value 0.

```
In [21]: print('sparsity:', 1.0 - messages_bow.nnz / (messages_bow.shape[0]* messages_b
         ow.shape[1]))
```

```
sparsity: 0.9986699797000932
```

Next, lets see what the bow array looks like if we convert it to a "dense" array and print it out. Lots of 0s right? We can calculate the storage required by using `sys.getsizeof(python_array)` so please add that call to the following cell. The numpy array requires a different size measurement because it stores the array outside of Python, so you can use `numpy_array.data.nbytes` to find its size

```
In [22]: messages_array = messages_bow.toarray()
         print(messages_array)
```

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

Memory Usage Calculation:

The below cell is responsible for outputting the number of bytes contained in the **“dense” array** representation of the sparse matrix **messages_bow**.

```
In [23]: sys.getsizeof(messages_array)
```

```
Out[23]: 490958032
```

The below cell is responsible for outputting the number of bytes contained in the **sparse matrix** that was created, **messages_bow**.

```
In [24]: messages_bow.data.nbytes
```

```
Out[24]: 652984
```

Term weighting and normalization can be done with TF-IDF (http://en.wikipedia.org/wiki/Tf%E2%80%93idf), using scikit-learn's `TfidfTransformer` , and we can apply it to the message we used above.

In [25]:
```
tfidf_transformer = TfidfTransformer().fit(messages_bow)
tfidf4 = tfidf_transformer.transform(bow4)
print(tfidf4)
```

```
  (0, 10054)	0.22510385070095637
  (0, 9589)	0.1955442748962185
  (0, 9280)	0.49597495370832545
  (0, 7137)	0.4269339327922034
  (0, 6243)	0.3100112284407115
  (0, 6219)	0.2913528957227454
  (0, 5363)	0.2860779240943588
  (0, 4762)	0.25892595706356525
  (0, 4189)	0.391088549792437
```

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

In [26]:
```
messages_tfidf = tfidf_transformer.transform(messages_bow)
print(messages_tfidf.shape)
```

```
(5574, 11010)
```

Part 4: Training a model, detecting spam

With messages represented as vectors, we can finally train our spam/ham classifier.

We'll be using scikit-learn here, choosing the Naive Bayes (http://en.wikipedia.org/wiki/Naive_Bayes_classifier) classifier:

In [27]:
```
spam_detector = MultinomialNB().fit(messages_tfidf, messages['label'])
```

Let's try classifying our single random message:

In [28]:
```
print('predicted:', spam_detector.predict(tfidf4)[0])
print('expected:', messages.label[3])
```

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

Hooray!

A natural question is to ask, how many messages do we classify correctly overall? The following cell will calculate this for us...

In [29]:
```
all_predictions = spam_detector.predict(messages_tfidf)
print('accuracy', accuracy_score(messages['label'], all_predictions))
```

```
accuracy 0.9721923214926445
```

There are quite a few possible metrics for evaluating model performance. Which one is the most suitable depends on the task. For example, the cost of mispredicting "spam" as "ham" is probably much lower than mispredicting "ham" as "spam". Differences between errors can be illuminated using metrics other than accuracy, so in the following cell, and in the cells below, you should use sklearn to calculate recall and precision in addition to accuracy. Please include statements about what you can interpret from these results

Calculating Recall

The below cell is responsible for checking recall, with a focus on the correct identification of messages being labeled as **spam**. What the output of the below cell conveys is what proportion of messages that were actually **spam** were identified as **spam**.

In [30]:
```
from sklearn.metrics import recall_score
print('recall score', recall_score(all_predictions, messages['label'], pos_lab
el='spam'))
```

```
recall score 1.0
```

Calculating Precision

The below cell is responsible for checking precision, with a focus on the correct identification of messages being labeled as **spam**. What the output of the below cell conveys is what proportion of messages that were identified as **spam** were actually **spam**.

In [31]:
```
from sklearn.metrics import precision_score
print('precision score', precision_score(all_predictions, messages['label'], p
os_label='spam'))
```

```
precision score 0.7925033467202142
```

Part 5: Let's get realistic

In the above "evaluation", we committed a cardinal sin. For simplicity of demonstration, we evaluated accuracy on the same data we used for training. **Never 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. This is exactly like memorizing the exact answers for an exam without understanding the underlying material!

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 -- thanks to this process, we make sure we are not "cheating", and that our final evaluation on test data is representative of true predictive performance.

The following code splits the dataset into a training and testing set.

Using 20% of Data for Testing:

In [32]:
```
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))
```

```
4459 1115 5574
```

So, as requested, the test size is 20% of the entire dataset.

Next, lets set up our split datasets to be ready to be used by the Bayes model for training and prediction...

In [33]:
```
train_messages_bow = bow_transformer.transform(msg_train)
train_tfidf_transformer = TfidfTransformer().fit(train_messages_bow)
train_messages_tfidf = train_tfidf_transformer.transform(train_messages_bow)
test_messages_bow = bow_transformer.transform(msg_test)
test_tfidf_transformer = TfidfTransformer().fit(test_messages_bow)
test_messages_tfidf = test_tfidf_transformer.transform(test_messages_bow)
```

We can train a new Naive Bayes classifier with only the training data, and test it with the test data, and our accuracy should drop. In this cell answer: why?

Following cell modified by Nigel Nelson:

By training with one subset of the complete data, and testing with another subset of the test data, the accuracy of our model should drop. The reason for this is that in our previous model, the model was trained on the complete data set, and testing on the data from the same data it was trained on. What this allows is the model to essentially memorize the correct identification of certain messages instead of finding patterns in order to correctly predict spam. So, by testing on different data than the training data, this accuracy is a true representation of how accurate our model is at identifying the patterns of spam messages. This means that this accuracy should be lower than the previous one because it is much easier to recall information that has already been provided, than it is to predict information that is being seen for the first time.

```
In [34]: split_spam_detector = MultinomialNB().fit(train_messages_tfidf, label_train)
         test_predictions = split_spam_detector.predict(test_messages_tfidf)
         print('updated accuracy', accuracy_score(label_test, test_predictions))
```

```
updated accuracy 0.95695067264574
```

Next, re-run this experiment changing the test size to a different value (in the subsequent cells of this part) and develop an explanation for the results (it should be different than your accuracy value)

*Below **4** cells modified by Nigel Nelson:*

Using 40% of Data for Training

The below cells are responsible for mirroring the functionality of the above section that uses 20% of the data for testing, except below 40% of the data is used for testing in order to see the effect had on the accuracy of the model.

```
In [35]: msg_train40, msg_test40, label_train40, label_test40 = \
             train_test_split(messages['message'], messages['label'], test_size=0.4)

         print(len(msg_train40), len(msg_test40), len(msg_train40) + len(msg_test40))
```

```
3344 2230 5574
```

```
In [36]: train_messages_bow40 = bow_transformer.transform(msg_train40)
         train_tfidf_transformer40 = TfidfTransformer().fit(train_messages_bow40)
         train_messages_tfidf40 = train_tfidf_transformer40.transform(train_messages_bo
         w40)
         test_messages_bow40 = bow_transformer.transform(msg_test40)
         test_tfidf_transformer40 = TfidfTransformer().fit(test_messages_bow40)
         test_messages_tfidf40 = test_tfidf_transformer40.transform(test_messages_bow40
         )
```

In [37]:
```
split_spam_detector40 = MultinomialNB().fit(train_messages_tfidf40, label_trai
n40)
test_predictions40 = split_spam_detector40.predict(test_messages_tfidf40)
print('updated accuracy', accuracy_score(label_test40, test_predictions40))
```

```
updated accuracy 0.9452914798206278
```

Analysis of Accuracy Difference

The first approach where the test data accounted for 20% of the total data, that model resulted in an approximate 2% advantage over the model that had 40% of its total data used for testing. This result makes sense due to the fact that the extra 20% for the second tested model had to come from somewhere, and that source would be the training data. When training data is removed, the model sees less examples of what to look for when screening for spam, and due to this, the model has less of a knowledge base to base it predictions on, resulting in a decline in accuracy. Essentially, this trend mirrors the saying *practice makes perfect*, just as a student's grades increase as they study more, the model's accuracy will increase as it is able to be trained on more data

---

Part 6: Next Steps

In the following cells you should make some changes to the dataset (cast to lowercase, remove numbers, remove non-words, add content, etc) to sufficiently change the sparsity percentage. The number of columns in your bag of words model should be significantly smaller. The goal of this is see the size comparison in the non-compressed version of the matrix (`toarray`) vs the sparse representation as the size of the data changes. If we didn't have a sparse representation, our ability to use a BOW model would be very limiting...

Run the experiments again to assess the accuracy of your new dataset and compare it with your previous results. You should make arguments about what caused the changes and why they make sense. Calculate and compare the storage requirements of the non-sparse and sparse representations, and argue how using sparse matricies can enable better accuracy.

Remainder of Notebook modified by Nigel Nelson

Importing nltk

The below cell is responsible for importing nltk, a leading platform for building Python programs to work with human language data.

In [38]:
```
import nltk
```

Creating a Set of English Words

The below cell is responsible for creating a set **words** that is comprised of every english word that **nltk** has to offer.

In [39]:
```
words = set(nltk.corpus.words.words())
```

Checking if a token is an English Word

The below cell is responsible for checking if the tokens that appears in a string of tokens **data**, are real words according to **words**, if they are they are concatenated to a new string of tokens that is returned.

In [40]:
```
def words_only(data, words):
    s = ''
    for token in data.split():
        if token in words:
            s += ' ' + token
    return s
```

Converting to Lowercase

The below cell is responsible for creating a new column **lowercase** in the dataframe **messages** that contains the individual messages with each character in lowercase.

In [41]:
```
messages['lowercase'] = messages.message.apply(lambda s: s.lower())
messages.head()
```

Out[41]:

	label	message	length	lowercase
0	ham	Go until jurong point, crazy.. Available only ...	111	go until jurong point, crazy.. available only ...
1	ham	Ok lar... Joking wif u oni...	29	ok lar... joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	155	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...	49	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...	61	nah i don't think he goes to usf, he lives aro...

Removing Non-Letters

The below cell is responsible for creating a new column **letters_only** in the dataframe **messages** which only contains the characters from **lowercase** that are alphabetical or instances of a space.

In [42]:
```
messages['letters_only'] = messages['lowercase'].apply(lambda s: re.sub(r'[^a-z\\d\s]','',s))
messages.head()
```

Out[42]:

	label	message	length	lowercase	letters_only
0	ham	Go until jurong point, crazy.. Available only ...	111	go until jurong point, crazy.. available only ...	go until jurong point crazy available only in ...
1	ham	Ok lar... Joking wif u oni...	29	ok lar... joking wif u oni...	ok lar joking wif u oni
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	155	free entry in 2 a wkly comp to win fa cup fina...	free entry in a wkly comp to win fa cup final...
3	ham	U dun say so early hor... U c already then say...	49	u dun say so early hor... u c already then say...	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...	61	nah i don't think he goes to usf, he lives aro...	nah i dont think he goes to usf he lives aroun...

Removing Non-English words

The below cell is responsible for creating the column **english_words_only** which is a result of applying the function **words_only** to the column **letters_only**.

In [43]:
```
messages['english_words_only'] = messages['letters_only'].apply(lambda s: words_only(s, words))
messages.head()
```

Out[43]:

	label	message	length	lowercase	letters_only	english_words_only
0	ham	Go until jurong point, crazy.. Available only ...	111	go until jurong point, crazy.. available only ...	go until jurong point crazy available only in ...	go until point crazy available only in n grea...
1	ham	Ok lar... Joking wif u oni...	29	ok lar... joking wif u oni...	ok lar joking wif u oni	lar u
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	155	free entry in 2 a wkly comp to win fa cup fina...	free entry in a wkly comp to win fa cup final...	free entry in a to win fa cup final st may te...
3	ham	U dun say so early hor... U c already then say...	49	u dun say so early hor... u c already then say...	u dun say so early hor u c already then say	u dun say so early u c already then say
4	ham	Nah I don't think he goes to usf, he lives aro...	61	nah i don't think he goes to usf, he lives aro...	nah i dont think he goes to usf he lives aroun...	i dont think he goes to he around here though

Creating Updated Data Vectors

The below cell is responsible for converting each message, represented as a list of tokens (lemmas) above, into a vector that machine learning models can understand.

Doing that requires essentially three steps, in the bag-of-words model:

1. counting how many times does a word occur in each message (term frequency)
2. weighting the counts, so that frequent tokens get lower weight (inverse document frequency)
3. normalizing the vectors to unit length, to abstract from the original text length

Following this, the number of vocabulary words in the updated vectors are output to display differences with the first instance of data vectors.

In [44]:
```
bow_transformer2 = CountVectorizer(analyzer=split_into_lemmas).fit(messages['e
nglish_words_only'])
print(len(bow_transformer2.vocabulary_))
```

```
3721
```

Creating Updated Sparse Matrix

The below cell is responsible for creating bag-of-words counts for the entire SMS corpus, represented by a sparse matrix (generated using `bow_transformer2.transform()` on the column **message**, which is the original message collected).

In [45]:
```
messages_bow2 = bow_transformer2.transform(messages.message)
```

Dimensions of Updated Sparse Matrix:

The below cell returns a tuple of the dimensions of the created updated sparse matrix, where the first value is the number rows of the matrix, which are composed of the individual messages, and the second value is the number of columns, which is the individual unique tokens found in the messages

In [46]:
```
messages_bow2.shape
```

Out[46]:
```
(5574, 3721)
```

Updated Calculation of Sparsity:

The below cell calcualtes the sparsity of the created sparse matrix, or the percentage of the matrix that contains the value 0. This is done by dividing the number of non-zero entries by the number of elements in the sparse matrix, accomplished by multiplying the number of rows by the number of columns. That number is then subtracted from 1.0 to give the percentage of the matrix that contains the value 0.

In [47]:
```
print('sparsity:', 1.0 - messages_bow2.nnz / (messages_bow2.shape[0]* messages
_bow2.shape[1]))
```

```
sparsity: 0.9974913761988778
```

Updated Memory Benchmarking

The below cell is responsible for creating a **dense** python array from the sparse matrix **messages_bow2**

In [48]:
```
messages_array2 = messages_bow2.toarray()
print(messages_array)
```

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

The below cell is responsible for outputting the number of bytes contained in the **“dense” array** representation of the sparse matrix **messages_bow2**.

In [49]:
```
sys.getsizeof(messages_array2)
```

Out[49]:
```
165926944
```

The below cell is responsible for outputting the number of bytes contained in the **sparse matrix** that was created, **messages_bow2**.

In [50]:
```
messages_bow2.data.nbytes
```

Out[50]:
```
416248
```

Updated Term Weighting

Updating Term weighting and normalization can be done with [TF-IDF (http://en.wikipedia.org/wiki/Tf%E2%80%93idf)](http://en.wikipedia.org/wiki/Tf%E2%80%93idf), using scikit-learn's `TfidfTransformer`

In [51]:
```
tfidf_transformer2 = TfidfTransformer().fit(messages_bow2)
```

Updated TF-IDF Corpus

The below cell is responsible for transforming the entire updated bag-of-words corpus into TF-IDF corpus at once, whose dimensions are then output.

In [52]:
```
messages_tfidf2 = tfidf_transformer2.transform(messages_bow2)
print(messages_tfidf2.shape)
```

```
(5574, 3721)
```

Using 20% of Data for Updated Testing:

The below cell is responsible for subsetting the complete data set into testing and training sets of data, where 20% of the data is reserved for testing.

In [53]:
```
msg_train2, msg_test2, label_train2, label_test2 = \
    train_test_split(messages['message'], messages['label'], test_size=0.2)

print(len(msg_train2), len(msg_test2), len(msg_train2) + len(msg_test2))
```

```
4459 1115 5574
```

Updated Split Datasets

The below cell is responsible for splitting the datasets to be ready to be used by the Bayes model for training and prediction.

In [54]:
```
train_messages_bow2 = bow_transformer2.transform(msg_train2)
train_tfidf_transformer2 = TfidfTransformer().fit(train_messages_bow2)
train_messages_tfidf2 = train_tfidf_transformer2.transform(train_messages_bow2
)
test_messages_bow2 = bow_transformer2.transform(msg_test2)
test_tfidf_transformer2 = TfidfTransformer().fit(test_messages_bow2)
test_messages_tfidf2 = test_tfidf_transformer2.transform(test_messages_bow2)
```

Updated Accuracy of Model

The below cell is responsible for testing the predictions of the updated model, and then outputting the updated accuracy of the model.

In [55]:
```
split_spam_detector2 = MultinomialNB().fit(train_messages_tfidf2, label_train2
)
test_predictions2 = split_spam_detector2.predict(test_messages_tfidf2)
print('updated accuracy', accuracy_score(label_test2, test_predictions2))
```

```
updated accuracy 0.9354260089686098
```

Reflection on Accuracy Changes

When the accuracy of the original model is compared to the updated model, there is approximately a **2.24%** advantage in accuracy for the original model. The reason that this difference may exist is likely due to the token cleaning that was utilized for the updated model. This is because both models were tested on the un-edited SMS messages, yet the updated model was built on these SMS messages after all messages were set to lowercase, non-words were removed, and any non-alphabetical character was removed besides spaces. As such, the updated model did not have the ability to search for correlations and patterns that could be contained in the capitalization, the non-alphabetical characters, and the non-words found in the SMS messages. Due to a lack of representation for the tokens as they appeared in the testing data set, the updated model's accuracy suffered.

Relfection on Sparsity vs. Accuracy

The original model and the updated model were not only different in their accuracies, they were also different in their sparsity's. The original model was approximately **.12%** more sparse. I believe that a predictor of accuracy may be the sparsity of the matrix, where as sparsity increases so does accuracy. The reason for this is that if for the same dataset, one matrix is sparser than the other in a bag of words approach, that means that there are more vocabulary words being tested for in messages. As such, there are more data points to use in order to build stronger correlations between certain tokens in order to reach more certainty in positive identifications.

Conclusion

The purpose of this notebook was to begin a guided exploration into machine learning using sparse matrixes. Specifically this notebook set out to create a machine learning model that was trained on SMS messages so that it could identify if messages were spam or not using a bag of words approach. It was discovered in the notebook in a bag of words approach for a large set of text data, the sparse matrix that results often contains a decimal of a percent of non-zero values. Due to this massive of amount of non-zero values being stored, it was discovered that when these sparse matrixes were converted to a *dense* array representation, sparse matrixes stored approximately **750 times** less bytes for the examples ran in this notebook. In addition, it was learned that when creating a machine learning model, you should always have subdivided training data and testing data, as this ensures that your accuracy is a legitimate representation of how effectively the model *learned* to identify positive matches. However, the higher of a percentage used for testing data the less training data is used for the model, and as such the accuracy of the model drops. Lastly, it was discovered that by creating a model based off of cleaned input data, the resulting sparse matrix is less sparse than its non-cleaned counterpart, in addition to being less accurate. From these results the conclusion was made that in terms of a bag of words approach, the sparser the matrix created, the greater accuracy the model will have. This was hypothesized to be due to the fact that a sparser matrix has more columns of tokens, which means there are more possibilities for correlations and patterns to drawn from, which creates greater certainty when making positive identifications. This increase in accuracy associated with an increase in sparsity is made possible by sparse matrixes. As mentioned above, sparse matrixes reduced the amount of bytes being stored by 450 times, if it wasn’t for this significantly smaller storage capacity, it would not be feasible to use massively sparse traditional matrixes in order to create accuracy gains as it would require an unreasonable amount of storage.