Text classification using the Bag Of Words Approach with NLTK and Scikit Learn

medium.com/swlh/text-classification-using-the-bag-of-words-approach-with-nltk-and-scikit-learn-9a731e5c4e2f

May 7, 2020

Charles Rajendran

May 6, 2020

Text Classification is an important area in machine learning, there is a wide range of applications that depends on text classification. Let's take some examples.

Spam Filtering: This is a very matured field in text classification, filtering which email to show in the inbox and which email to put in the spam.

Sentiment Analysis: This is a classification task which will classify people's opinion expressed in a piece of text.

Intention Mining: Finding the future decision of a person based on the text.

These are some sub-areas in the field of Text Classification, but there are much more use cases than these.

Machine Learning algorithms work very well with numbers, but when it comes to text, we have to do some preprocessing to make our model predict well. Let's see about these steps practically with an SMS spam filtering program.

Step 1: Import the data

```
import pandas as pddataset = pd.read_csv('data.csv', encoding='ISO-8859-1');
```

In this example, I have used a dataset from Kaggle and imported it using a popular python library for data analysis called pandas. Pandas will allow us to import the data in CSV and stored it into a pandas specific data structure called Data Frame. Data in data frames can be easily accessible, and we can perform many operations with it as well.

If you see *read_csv* function, we have passed a parameter for encoding, this is because our data set contain a certain character that is not supported by the default codec of pandas *read_csv* function. Because in SMS we might send emoji, non-english words and etc. To know more about encoding in python.

Step 2: Preprocessing the data

This is a very important step in text classification since machine learning algorithms are not very good at working with text comparing to numbers. So we have to remove every unwanted stuff in the text before putting the data into the model. Let's talk about some common preprocessing steps we can do to the text.

a. Remove none alphabetic characters

When it comes to text classification, we use words as the features, so it's important to remove unwanted characters such as numbers and punctuation marks. (emoji will represent some meaning especially when it comes to sentiment analysis, but for the scope of this article I will remove those as well.)

```
import resms = re.sub('[^A-Za-z]', ' ', sms
```

Let's come to the code, to remove the none alphabetic characters I used regular expression module of python. The *sub* method will take three parameters.

1. Pattern — The above pattern says any character other than A-Z and a-z.
2. Replacement — Any character matched with the above pattern will be replaced by the value in the second parameter.
3. Content

b. Make the word lower case

Otherwise *'Go'* and *'go'* will be treated as two different words. Also, later on, we will remove stop words from the text, words in the stop word list are in lowercase so checking the existence of the word in that list is easy.

```
sms = sms.lower()
```

c. Remove the stop words

Stop words in text classification are words that don't have any impact on deciding the class of the text. For example words like *the, we, a, will* and etc.

Let's come to the code again, our approach to removing stop words is, we compare each word in SMS against the stop word list, if the word exists in the list of stop word then we ignore those.

First, we need to split the words in the SMS, for this, I have used a tokenizer available in the nltk library.

```
import nltk

nltk.download('punkt')
from nltk.tokenize import word_tokenize

tokenized_sms = word_tokenize(sms)
```

Let’s go through the code.

There are a number of datasets available in nltk, such as movie review data, names data and etc. The **punkt** dataset is one of them and it’s required to train the tokenizers in nltk.

This tokenizer will tokenize the text, and create a list of words.

Since we got the list of words, it’s time to remove the stop words in the list words.

```
nltk.download('stopwords')
from nltk.corpus import stopwords

for word in tokenized_sms:    if word in stopwords.words('english'):
tokenized_sms.remove(word)
```

d. Stemming

Stemming is the process of finding the base word. Let’s take an example, the base word for words runnable, running is run. This is very important because in bag of word model the words appeared more frequently are used as the features for the classifier, therefore we have to remove such variations of the same word.

For stemming, again we going to use a model in nltk called .

```
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()

for i in range(len(tokenized_sms)):    tokenized_sms[i] =
stemmer.stem(tokenized_sms[i])
```

e. Spell correction

Mostly SMS text will not have the correct spellings. So using a spell correction will be handy. For spell correction, I have used a python library called autocorrect.

```
from autocorrect import spelltokenized_sms[i] = stemmer.stem(spell(tokenized_sms[i]))
```

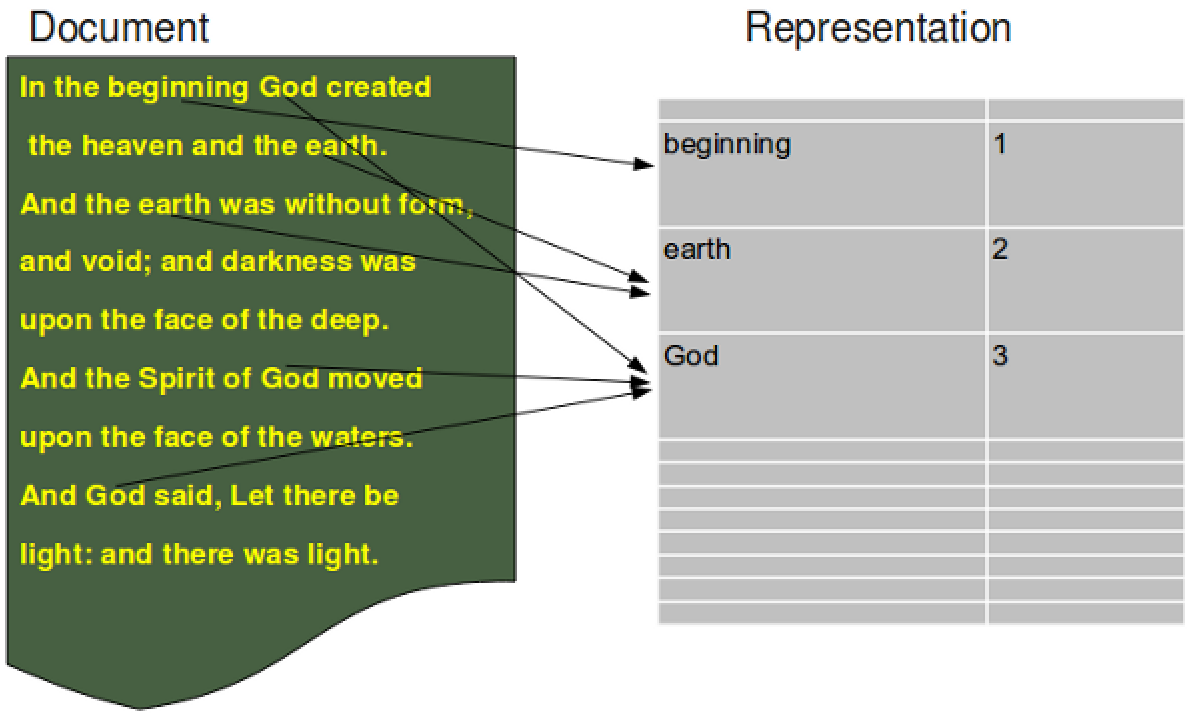
After all these preprocessing work, we can create the text with the list of words in the tokenized_sms list.

```
sms_text = " ".join(tokenized_sms)
```

Here I have used these preprocessing to one SMS, we have to do the same thing to all the SMS in the dataset.

Step 3: Create the model and train

As I said before, in this article I am going to use the bag of word approach to classify. So let's understand the bag of word model.


Document
In the beginning God created
the heaven and the earth.
And the earth was without form,
and void; and darkness was
upon the face of the deep.
And the Spirit of God moved
upon the face of the waters.
And God said, Let there be
light: and there was light.
Representation
beginning
1
earth
2
God
3

In the bag of words approach, we will take all the words in every SMS, then count the number of occurrences of each word. After finding the number of occurrences of each word, we will choose a certain number of words that appeared more often than other words. Let's say we choose the most frequent 1000 words. Then these 1000 words are the features for our classification problem. Then when it comes to classification, the classifier, will check what are the words in the feature set appeared in the new SMS text and based on that it will decide the class of new SMS.

> Let's take an example to understand this more clearly.
>
> SMS 1: Pay your connection bill of 2000. **class — not spam**
>
> SMS 2: Win 20000 by participating in the lottery. **class — spam**
>
> SMS 3: Win 200 rupees by participating in this competition at gamer.com. **class — spam**
>
> SMS 4: I will come tomorrow. **class — not spam**

If you see the above example, words like *Win, Participate* have appeared more times than other words, therefore these are the words that will be selected as features. Also if you notice these words have appeared more when the SMS is spam, therefore, if a new SMS containing such words, then most probably this SMS will be in the spam class. This is how the bag of words model works.

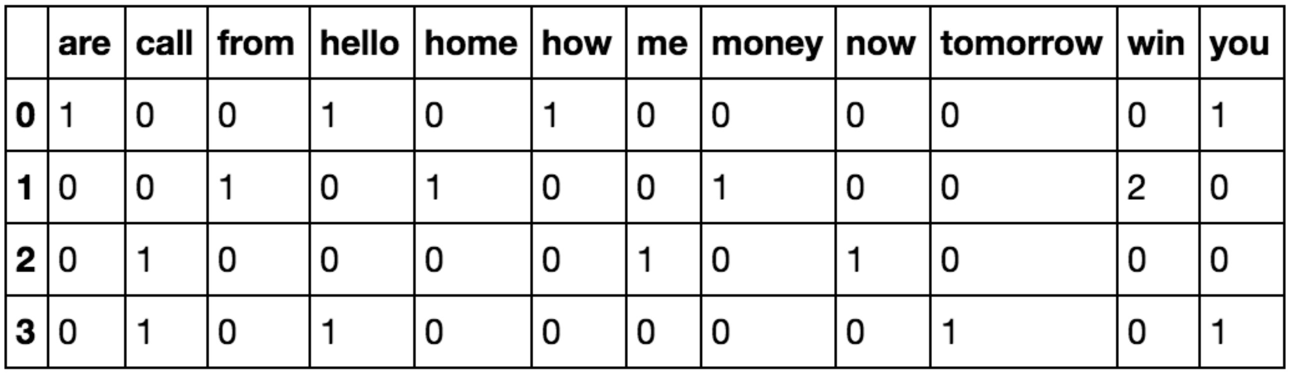
Let's move to the code.

Speaking about the bag of words, it seems like, we have tons of work to do, to train the model, like splitting the words in the corpus (dataset), Counting the frequency of words, selecting most frequent words as features. But the good news is, we don't have to worry about this, because *sklearn* provide us with an excellent function called .

```
from sklearn.feature_extraction.text import CountVectorizermatrix =
CountVectorizer(max_features=1000)X = matrix.fit_transform(data).toarray()
```

As you can see in the above code the *CountVectorizer* method accepts many parameters, such as *max_features*, *ngram* and so on, but here we are only interested in giving the number of most frequent words to choose as features.

In the above code the *CountVectorizer's fit transform* method will create a matrix, the rows in the matrix are SMS and the columns are the feature words, each cell in the matrix will denote the number of times a word appeared in the SMS. Such as in the below image.



	are	call	from	hello	home	how	me	money	now	tomorrow	win	you
0	1	0	0	1	0	1	0	0	0	0	0	1
1	0	0	1	0	1	0	0	1	0	0	2	0
2	0	1	0	0	0	0	1	0	1	0	0	0
3	0	1	0	1	0	0	0	0	0	1	0	1

We have the *X* variable now for the classifier, now we need the *y* variable, *y* means the target/class whether it is a spam SMS or a none spam SMS. This is available in the pandas data frame object which we imported initially. After that, we will split the dataset into train and test set.

```
y = dataset.iloc[:, 0]

from sklearn.model_selection import train_test_splitX_train, X_test, y_train, y_test
= train_test_split(X, y)
```

Step 4: Use a model for classification and find accuracy.

In this article, I have used Gaussian Naive Bayes Model to predict the class. Naive Bayes is one classification algorithm that works well with text data, so I have used that here, Decision Tree, Random Forest are some other algorithms that work well with text data. Then the rest is quite obvious, predicting the classes for the test set and calculating the accuracy by comparing the predicted results with the actual test set result.

```
# Naive Bayes
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, y_train)

# Predict Class
y_pred = classifier.predict(X_test)

from sklearn.metrics import accuracy_scoreaccuracy = accuracy_score(y_test, y_pred)
```

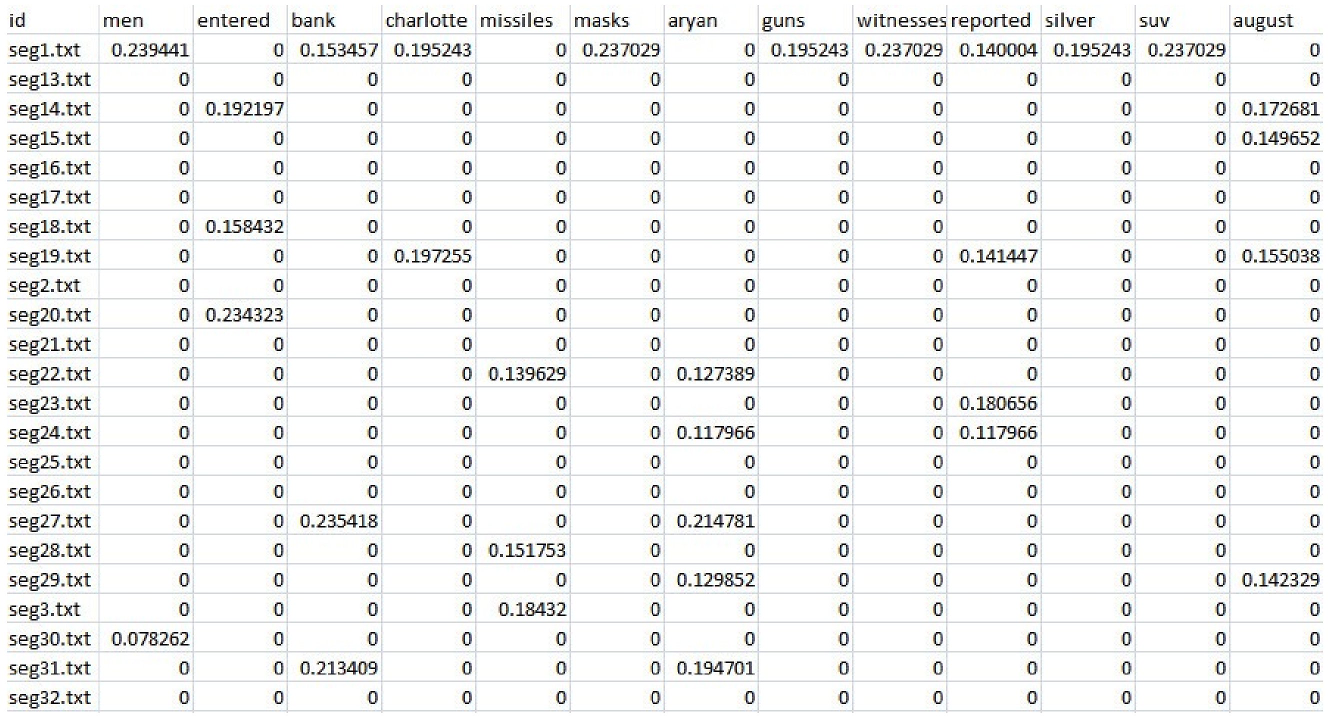
I have obtained an accuracy of 80% with this classifier, but there are a number of different ways we can improve this, such as,

For the bag of words model here we have used words (unigram) as a feature set. This might be a problem in some cases, especially in sentiment analysis. For example *"This is not a good movie"* is a negative text, but if we select words as features, then it will tokenize the text by words, therefore this might get classified as a positive text since it has the word *"good"*. If it takes *"not good"* as a feature (bigram) then it will classify such cases correctly.

2. Using TF-IDF vectors or Vector Space Model over Count Vectors

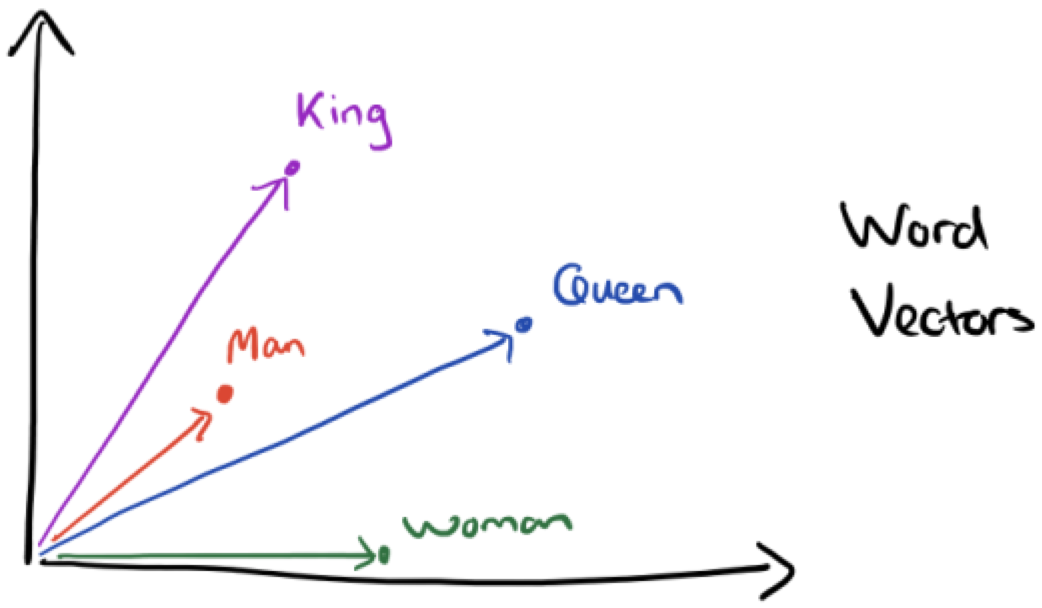
TF-IDF vectors — TF-IDF score represents the relative importance of a term in the document and the entire corpus.

```
TF(t) = (Number of times term t appears in a document) / (Total number of terms in
the document)IDF(t) = log_e(Total number of documents / Number of documents with term
t in it)
```



id	men	entered	bank	charlotte	missiles	masks	aryan	guns	witnesses	reported	silver	suv	august
seg1.txt	0.239441	0	0.153457	0.195243	0	0.237029	0	0.195243	0.237029	0.140004	0.195243	0.237029	0
seg13.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg14.txt	0	0.192197	0	0	0	0	0	0	0	0	0	0	0.172681
seg15.txt	0	0	0	0	0	0	0	0	0	0	0	0	0.149652
seg16.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg17.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg18.txt	0	0.158432	0	0	0	0	0	0	0	0	0	0	0
seg19.txt	0	0	0	0.197255	0	0	0	0	0	0.141447	0	0	0.155038
seg2.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg20.txt	0	0.234323	0	0	0	0	0	0	0	0	0	0	0
seg21.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg22.txt	0	0	0	0	0.139629	0	0.127389	0	0	0	0	0	0
seg23.txt	0	0	0	0	0	0	0	0	0	0.180656	0	0	0
seg24.txt	0	0	0	0	0	0	0.117966	0	0	0.117966	0	0	0
seg25.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg26.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg27.txt	0	0	0.235418	0	0	0	0.214781	0	0	0	0	0	0
seg28.txt	0	0	0	0	0.151753	0	0	0	0	0	0	0	0
seg29.txt	0	0	0	0	0	0	0.129852	0	0	0	0	0	0.142329
seg3.txt	0	0	0	0	0.18432	0	0	0	0	0	0	0	0
seg30.txt	0.078262	0	0	0	0	0	0	0	0	0	0	0	0
seg31.txt	0	0	0.213409	0	0	0	0.194701	0	0	0	0	0	0
seg32.txt	0	0	0	0	0	0	0	0	0	0	0	0	0

It is a way of representing document as vectors. There are pre-trained models available for word embedding. Such as Word2Vec and Glove.



That's it, hope you grab some knowledge out of this article 😉.

All my code is available in Github and feel free to follow me on Github 😉.