Multinomial and Bernoulli Naive Bayes

For understanding Multinomial and Bernoulli Naive Bayes, we will take a few sentences and classify them in two different classes. Each sentence will represent one document. In real world examples, every sentence could be a document, such as a mail, or a news article, a book review, a tweet etc.

The analysis and mathematics involved doesn’t depend on the type of document we use. Therefore we have chosen a set of small sentences to demonstrate the calculation involved and to drive in the concept.

Let us first look at the sentences and their classes. We have kept these sentences in file example_train.csv. Test sentences have been put in the file example_test.csv.

In [13]:
```
import numpy as np
import pandas as pd
import sklearn

docs = pd.read_csv('example_train.csv')
#text in column 1, classifier in column 2.
docs
```

Out[13]:

	Document	Class
0	Upgrad is a great educational institution.	education
1	Educational greatness depends on ethics	education
2	A story of great ethics and educational greatness	education
3	Sholey is a great cinema	cinema
4	good movie depends on good story	cinema

So as you can see there are 5 documents (sentences) , 3 are of "education" class and 2 are of "cinema" class.

In [2]:
```
# convert label to a numerical variable
docs['Class'] = docs.Class.map({'cinema':0, 'education':1})
docs
```

Out[2]:

	Document	Class
0	Upgrad is a great educational institution.	1
1	Educational greatness depends on ethics	1
2	A story of great ethics and educational greatness	1
3	Sholey is a great cinema	0
4	good movie depends on good story	0

In [4]:
```
numpy_array = docs.as_matrix()
X = numpy_array[:,0]
Y = numpy_array[:,1]
Y = Y.astype('int')
print("X")
print(X)
print("Y")
print(Y)
```

```
X
['Upgrad is a great educational institution.'
 'Educational greatness depends on ethics'
 'A story of great ethics and educational greatness'
 'Sholey is a great cinema' 'good movie depends on good story']
Y
[1 1 1 0 0]
```

Imagine breaking X in individual words and putting them all in a bag. Then we pick all the unique words from the bag one by one and make a dictionary of unique words.

This is called **vectorization of words**. We have the class `CountVectorizer()` in scikit learn to vectorize the words. Let us first see it in action before explaining it further.

In [5]:
```
# create an object of CountVectorizer() class
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer( )
```

Here `vec` is an object of class `CountVectorizer()`. This has a method called `fit()` which converts a corpus of documents into a vector of unique words as shown below.

In [6]:
```
vec.fit(X)
vec.vocabulary_
```

Out[6]:
```
{'and': 0,
 'cinema': 1,
 'depends': 2,
 'educational': 3,
 'ethics': 4,
 'good': 5,
 'great': 6,
 'greatness': 7,
 'institution': 8,
 'is': 9,
 'movie': 10,
 'of': 11,
 'on': 12,
 'sholey': 13,
 'story': 14,
 'upgrad': 15}
```

`Countvectorizer()` has converted the documents into a set of unique words alphabetically sorted and indexed.

Stop Words

We can see a few trivial words such as 'and','is','of', etc. These words don't really make any difference in classyfying a document. These are called 'stop words'. So we would like to get rid of them.

We can remove them by passing a parameter stop_words='english' while instantiating `Countvectorizer()` as follows:

In [7]:
```
# removing the stop words
vec = CountVectorizer(stop_words='english' )
vec.fit(X)
vec.vocabulary_
```

Out[7]:
```
{'cinema': 0,
 'depends': 1,
 'educational': 2,
 'ethics': 3,
 'good': 4,
 'great': 5,
 'greatness': 6,
 'institution': 7,
 'movie': 8,
 'sholey': 9,
 'story': 10,
 'upgrad': 11}
```

Another way of printing the 'vocabulary':

In [10]:
```
# printing feature names
print(vec.get_feature_names())
print(len(vec.get_feature_names()))
```

```
['cinema', 'depends', 'educational', 'ethics', 'good', 'great', 'greatness',
'institution', 'movie', 'sholey', 'story', 'upgrad']
12
```

So our final dictionary is made of 12 words (after discarding the stop words). Now, to do classification, we need to represent all the documents with respect to these words in the form of features.

Every document will be converted into a *feature vector* representing presence of these words in that document. Let's convert each of our training documents in to a feature vector.

In [11]:
```
# another way of representing the features
X_transformed=vec.transform(X)
X_transformed
```

Out[11]:
```
<5x12 sparse matrix of type '<class 'numpy.int64'>'
        with 20 stored elements in Compressed Sparse Row format>
```

You can see X_tranformed is a 5 x 12 sparse matrix. It has 5 rows for each of our 5 documents and 12 columns each for one word of the dictionary which we just created. Let us print X_transformed.

In [25]:
```
print(X_transformed)
```

```
  (0, 2)	1
  (0, 5)	1
  (0, 7)	1
  (0, 11)	1
  (1, 1)	1
  (1, 2)	1
  (1, 3)	1
  (1, 6)	1
  (2, 2)	1
  (2, 3)	1
  (2, 5)	1
  (2, 6)	1
  (2, 10)	1
  (3, 0)	1
  (3, 5)	1
  (3, 9)	1
  (4, 1)	1
  (4, 4)	2
  (4, 8)	1
  (4, 10)	1
```

This representation can be understood as follows:

Consider first 4 rows of the output: (0,2), (0,5), (0,7) and (0,11). It says that the first document (index 0) has 7th , 2nd , 5th and 11th 'word' present in the document, and that they appear only once in the document- indicated by the right hand column entry.

Similarly, consider the entry (4,4) (third from bottom). It says that the fifth document has the fifth word present twice. Indeed, the 5th word('good') appears twice in the 5th document.

In real problems, you often work with large documents and vocabularies, and each document contains only a few words in the vocabulary. So it would be a waste of space to store the vocabulary in a typical dataframe, since most entries would be zero. Also, matrix products, additions etc. are much faster with sparse matrices. That's why we use sparse matrices to store the data.

Let us convert this sparse matrix into a more easily interpretable array:

In [17]:
```
# converting transformed matrix back to an array
# note the high number of zeros
X=X_transformed.toarray()
X
```

Out[17]:
```
array([[0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1],
       [0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0],
       [1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0],
       [0, 1, 0, 0, 2, 0, 0, 0, 1, 0, 1, 0]])
```

To make better sense of the dataset, let us examine the vocabulary and document-term matrix together in a pandas dataframe. The way to convert a matrix into a dataframe is `pd.DataFrame(matrix, columns=columns)`.

In [31]:
```
# converting matrix to dataframe
pd.DataFrame(X, columns=vec.get_feature_names())
```

Out[31]:

	cinema	depends	educational	ethics	good	great	greatness	institution	movie	shol
0	0	0	1	0	0	1	0	1	0	0
1	0	1	1	1	0	0	1	0	0	0
2	0	0	1	1	0	1	1	0	0	0
3	1	0	0	0	0	1	0	0	0	1
4	0	1	0	0	2	0	0	0	1	0

This table shows how many times a particular word occurs in document. In other words, this is a frequency table of the words.

A corpus of documents can thus be represented by a matrix with one row per document and one column per token (e.g. word) occurring in the corpus.

We call vectorization the general process of turning a collection of text documents into numerical feature vectors. This specific strategy (tokenization, counting and normalization) is called the "Bag of Words" representation. Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.

So, the 4 steps for vectorization are as follows

- Import
- Instantiate
- Fit
- Transform

Let us summarise all we have done till now:

- `vect.fit(train)` learns the vocabulary of the training data
- `vect.transform(train)` uses the fitted vocabulary to build a document-term matrix from the training data
- `vect.transform(test)` uses the fitted vocabulary to build a document-term matrix from the testing data (and ignores tokens it hasn't seen before)

In [37]:
```
test_docs = pd.read_csv('example_test.csv')
#text in column 1, classifier in column 2.
test_docs
```

Out[37]:

	Document	Class
0	very good educational institution	education

In [38]:
```
# convert label to a numerical variable
test_docs['Class'] = test_docs.Class.map({'cinema':0, 'education':1})
test_docs
```

Out[38]:

	Document	Class
0	very good educational institution	1

In [39]:
```
test_numpy_array = test_docs.as_matrix()
X_test = test_numpy_array[:,0]
Y_test = test_numpy_array[:,1]
Y_test = Y_test.astype('int')
print("X_test")
print(X_test)
print("Y_test")
print(Y_test)
```

```
X_test
['very good educational institution']
Y_test
[1]
```

In [40]:
```
X_test_transformed=vec.transform(X_test)
X_test_transformed
```

Out[40]:
```
<1x12 sparse matrix of type '<class 'numpy.int64'>'
	with 3 stored elements in Compressed Sparse Row format>
```

In [41]:
```
X_test=X_test_transformed.toarray()
X_test
```

Out[41]:
```
array([[0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]])
```

Multinomial Naive Bayes

In [48]:
```
# building a multinomial NB model
from sklearn.naive_bayes import MultinomialNB

# instantiate NB class
mnb=MultinomialNB()

# fitting the model on training data
mnb.fit(X,Y)

# predicting probabilities of test data
mnb.predict_proba(X_test)
```

Out[48]:
```
array([[ 0.32808399,  0.67191601]])
```

In [49]:
```
proba=mnb.predict_proba(X_test)
print("probability of test document belonging to class CINEMA" , proba[:,0])
print("probability of test document belonging to class EDUCATION" , proba[:,1
])
```

```
probability of test document belonging to class CINEMA [ 0.32808399]
probability of test document belonging to class EDUCATION [ 0.67191601]
```

In [50]:
```
pd.DataFrame(proba, columns=['Cinema','Education'])
```

Out[50]:

	Cinema	Education
0	0.328084	0.671916

Bernoulli Naive Bayes

In [51]:
```
from sklearn.naive_bayes import BernoulliNB

# instantiating bernoulli NB class
bnb=BernoulliNB()

# fitting the model
bnb.fit(X,Y)

# predicting probability of test data
bnb.predict_proba(X_test)
proba_bnb=bnb.predict_proba(X_test)
```

In [52]:
```
pd.DataFrame(proba_bnb, columns=['Cinema','Education'])
```

Out[52]:

	Cinema	Education
0	0.232637	0.767363

In the next sections, we will use Multinomial and Bernoulli Naive Bayes to solve an interesting real problem - classifying SMSes as spam or ham.