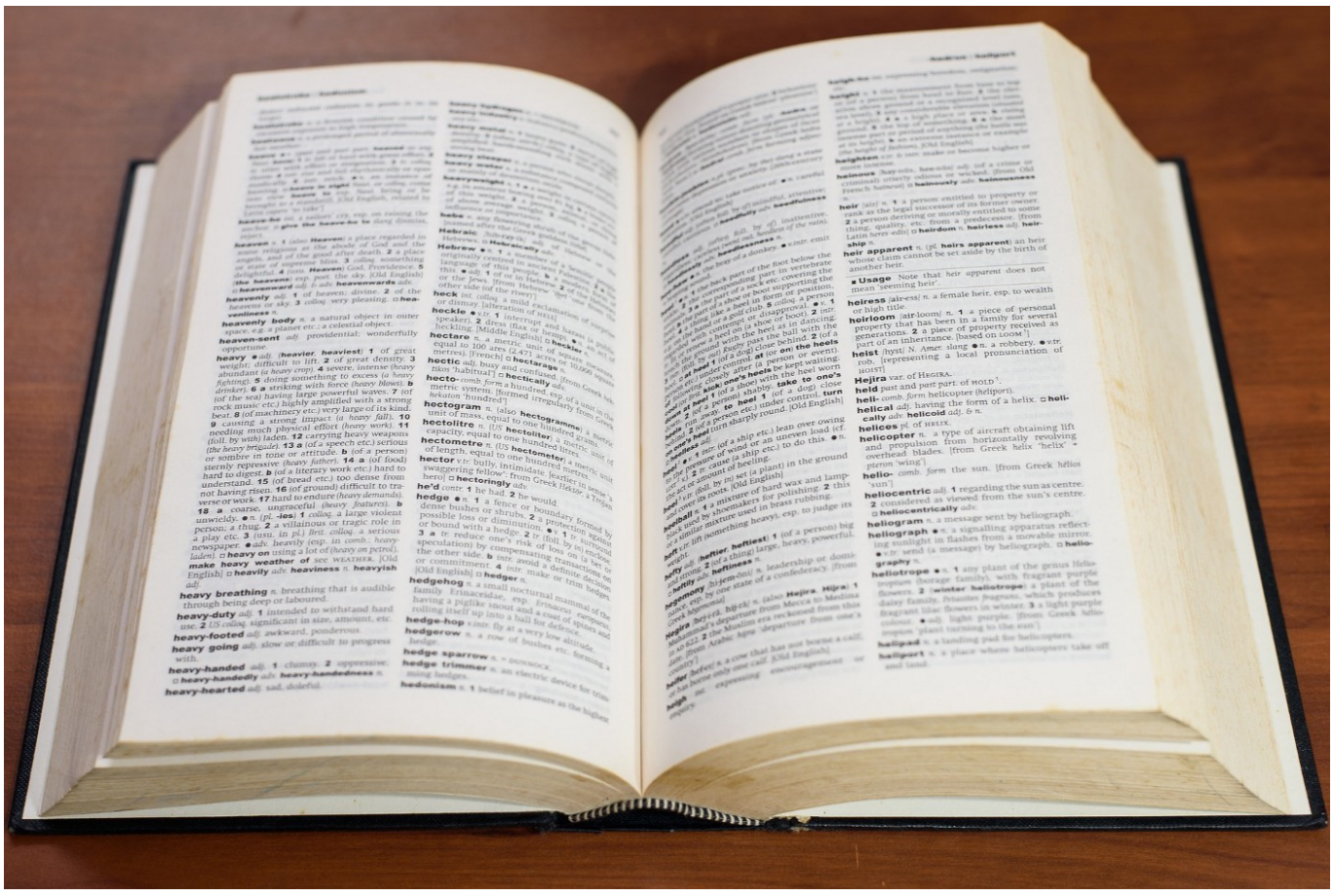
# *Multinomial Naive Bayes Classifier for Text Analysis (Python)*

*Syed Sadat Nazrul* *Apr 9, 2018 · 7 min read*



One of the most popular applications of machine learning is the analysis of categor
data, specifically text data. Issue is that, there are a ton of tutorials out there for nu

data but very little for texts. Considering how most of my past blogs on Machine Learning were based on Scikit-Learn, I decided to have some fun with this one by implementing the whole thing on my own.

In this blog, I will cover how you can implement a Multinomial Naive Bayes Classifie the 20 Newsgroups dataset. The 20 newsgroups dataset comprises around 18000 newsgroups posts on 20 topics split in two subsets: one for training (or developmer and the other one for testing (or for performance evaluation). The split between th train and test set is based upon a messages posted before and after a specific date

### *Libraries*

First, let us import the libraries needed for writing the implementation:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import operator
```

### *Class Distribution*

First, we calculate the fraction of documents in each class:

$$\pi_j = \frac{class_j}{\sum_{n=1}^{20} class_n}$$

```
#Training label
train_label = open('20news-bydate/matlab/train.label')

#pi is the fraction of each class
pi = {}

#Set a class index for each document as key
for i in range(1,21):
    pi[i] = 0

#Extract values from training labels
lines = train_label.readlines()

#Get total number of documents
total = len(lines)
```

```
#Count the occurence of each class
for line in lines:
    val = int(line.split()[0])
    pi[val] += 1

#Divide the count of each class by total documents
for key in pi:
    pi[key] /= total

print("Probability of each class:")
print("\n".join("{}: {}".format(k, v) for k, v in pi.items()))
```

```
Probability of each class:
1: 0.04259472890229834
2: 0.05155736977549028
3: 0.05075871860857219
4: 0.05208980388676901
5: 0.051024935664211554
6: 0.052533498979501284
7: 0.051646108794036735
8: 0.052533498979501284
9: 0.052888455053687104
10: 0.0527109770165942
11: 0.05306593309078002
12: 0.0527109770165942
13: 0.05244475996095483
14: 0.0527109770165942
15: 0.052622237998047744
16: 0.05315467210932647
17: 0.04836276510781791
18: 0.05004880646020055
19: 0.04117490460555506
20: 0.033365870973467035
```

## *Probability Distribution over Vocabulary*

Let’s first create the Pandas Dataframe

```
#Training data
train_data = open('20news-bydate/matlab/train.data')
df = pd.read_csv(train_data, delimiter=' ', names=['docIdx',
'wordIdx', 'count'])

#Training label
label = []
train_label = open('/home/sadat/Downloads/HW2_210/20news-
bydate/matlab/train.label')
lines = train_label.readlines()
for line in lines:
```

```
        label.append(int(line.split()[0]))

    #Increase label length to match docIdx
    docIdx = df['docIdx'].values
    i = 0
    new_label = []
    for index in range(len(docIdx)-1):
        new_label.append(label[i])
        if docIdx[index] != docIdx[index+1]:
            i += 1
    new_label.append(label[i]) #for-loop ignores last value

    #Add label column
    df['classIdx'] = new_label

    df.head()
```

| | docIdx | wordIdx | count | classIdx |
|---|---|---|---|---|
| **0** | 1 | 1 | 4 | 1 |
| **1** | 1 | 2 | 2 | 1 |
| **2** | 1 | 3 | 10 | 1 |
| **3** | 1 | 4 | 4 | 1 |
| **4** | 1 | 5 | 2 | 1 |

### *Probability of each word per class*

For calculating our probability, we will find the average of each word for a given cla

For class j and word i, the average is given by:

$$P(i|j) = \frac{word_{ij}}{word_j}$$

However, since some words will have 0 counts, we will perform a Laplace Smoothin with low $\alpha$:

$$P(i|j) = \frac{word_{ij} + \alpha}{word_j + |V| + 1}, \ \alpha = 0.001$$

where V is an array of all the words in the vocabulary

```
#Alpha value for smoothing
a = 0.001

#Calculate probability of each word based on class
pb_ij = df.groupby(['classIdx','wordIdx'])
pb_j = df.groupby(['classIdx'])
Pr =  (pb_ij['count'].sum() + a) / (pb_j['count'].sum() + 16689)

#Unstack series
Pr = Pr.unstack()

#Replace NaN or columns with 0 as word count with a/(count+|V|+1)
for c in range(1,21):
    Pr.loc[c,:] = Pr.loc[c,:].fillna(a/(pb_j['count'].sum()[c] +
16689))

#Convert to dictionary for greater speed
Pr_dict = Pr.to_dict()

Pr
```

| wordIdx | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | .. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| classIdx | | | | | | | | | | | |
| 1 | 7.855542e-05 | 0.000381 | 1.661627e-03 | 5.438638e-05 | 0.000495 | 0.000248 | 3.625960e-05 | 6.048302e-06 | 0.000205 | 8.459224e-04 | .. |
| 2 | 4.722740e-04 | 0.000464 | 7.871103e-09 | 1.338166e-04 | 0.000110 | 0.000457 | 7.871890e-05 | 4.723449e-05 | 0.001354 | 2.362118e-05 | .. |
| 3 | 1.023768e-04 | 0.000642 | 9.306135e-09 | 1.582136e-04 | 0.000195 | 0.000316 | 1.862158e-05 | 1.862158e-05 | 0.001340 | 9.306135e-09 | .. |
| 4 | 6.907239e-05 | 0.000268 | 8.632969e-09 | 8.632969e-09 | 0.000086 | 0.000414 | 1.727457e-05 | 8.641602e-06 | 0.000414 | 8.632969e-09 | .. |
| 5 | 5.833066e-05 | 0.000321 | 9.720157e-09 | 9.729877e-06 | 0.000010 | 0.000457 | 9.729877e-06 | 9.720157e-09 | 0.000457 | 9.720157e-09 | .. |
| 6 | 2.772348e-04 | 0.001309 | 5.898487e-09 | 4.659864e-04 | 0.000088 | 0.000307 | 1.238741e-04 | 1.770136e-05 | 0.001398 | 5.898487e-09 | .. |

## *Stop Words*

Stop words are words that show up a lot in every document (e.g. prepositions and pronouns).

```
#Common stop words from online
stop_words = [
"a", "about", "above", "across", "after", "afterwards",
"again", "all", "almost", "alone", "along", "already", "also",
"although", "always", "am", "among", "amongst", "amoungst", "amount",
"an", "and", "another", "any", "anyhow", "anyone", "anything",
"anyway", "anywhere", "are", "as", "at", "be", "became", "because",
"become","becomes", "becoming", "been", "before", "behind", "being",
"beside", "besides", "between", "beyond", "both", "but", "by","can",
"cannot", "cant", "could", "couldnt", "de", "describe", "do", "done",
"each", "eg", "either", "else", "enough", "etc", "even", "ever",
"every", "everyone", "everything", "everywhere", "except", "few",
"find","for","found", "four", "from", "further", "get", "give", "go",
"had", "has", "hasnt", "have", "he", "hence", "her", "here",
"hereafter", "hereby", "herein", "hereupon", "hers", "herself",
"him", "himself", "his", "how", "however", "i", "ie", "if", "in",
"indeed", "is", "it", "its", "itself", "keep", "least", "less",
"ltd", "made", "many", "may", "me", "meanwhile", "might", "mine",
"more", "moreover", "most", "mostly", "much", "must", "my", "myself",
"name", "namely", "neither", "never", "nevertheless", "next","no",
"nobody", "none", "noone", "nor", "not", "nothing", "now", "nowhere",
"of", "off", "often", "on", "once", "one", "only", "onto", "or",
"other", "others", "otherwise", "our", "ours", "ourselves", "out",
"over", "own", "part","perhaps", "please", "put", "rather", "re",
"same", "see", "seem", "seemed", "seeming", "seems", "she",
"should","since", "sincere","so", "some", "somehow", "someone",
"something", "sometime", "sometimes", "somewhere", "still", "such",
"take","than", "that", "the", "their", "them", "themselves", "then",
"thence", "there", "thereafter", "thereby", "therefore", "therein",
"thereupon", "these", "they",
"this", "those", "though", "through", "throughout",
"thru", "thus", "to", "together", "too", "toward", "towards",
"under", "until", "up", "upon", "us",
"very", "was", "we", "well", "were", "what", "whatever", "when",
"whence", "whenever", "where", "whereafter", "whereas", "whereby",
"wherein", "whereupon", "wherever", "whether", "which", "while",
"who", "whoever", "whom", "whose", "why", "will", "with",
"within", "without", "would", "yet", "you", "your", "yours",
"yourself", "yourselves"
]
```

Now, let’s create the vocabulary dataframe

```
vocab = open('vocabulary.txt')
vocab_df = pd.read_csv(vocab, names = ['word'])
vocab_df = vocab_df.reset_index()
vocab_df['index'] = vocab_df['index'].apply(lambda x: x+1)
vocab_df.head()
```

| | index | word |
|---|---|---|
| **0** | 1 | archive |
| **1** | 2 | name |
| **2** | 3 | atheism |
| **3** | 4 | resources |
| **4** | 5 | alt |

Getting the counts of each word in the vocabulary and setting stop words to 0:

```
#Index of all words
tot_list = set(vocab_df['index'])

#Index of good words
vocab_df = vocab_df[~vocab_df['word'].isin(stop_words)]
good_list = vocab_df['index'].tolist()
good_list = set(good_list)

#Index of stop words
bad_list = tot_list - good_list

#Set all stop words to 0
for bad in bad_list:
    for j in range(1,21):
        Pr_dict[j][bad] = a/(pb_j['count'].sum()[j] + 16689)
```

Multinomial Naive Bayes Classifier

Combining probability distribution of P with fraction of documents belonging to eac class.

For class j, word i at a word frequency of f:

$$Pr(j) \propto \pi_j \prod_{i=1}^{|V|} Pr(i|j)^{f_i}$$

In order to avoid underflow, we will use the sum of logs:

$$Pr(j) \propto \log(\pi_j \prod_{i=1}^{|V|} Pr(i|j)^{f_i})$$

$$Pr(j) \propto \log(\pi_j \prod_{i=1}^{|V|} Pr(i|j)^{f_i})$$

$$Pr(j) = \log \pi_j + \sum_{i=1}^{|V|} f_i \log(Pr(i|j))$$

One issue is that, if a word appears again, the probability of it appearing again goe
In order to smooth this, we take the log of the frequency:

$$Pr(j) = \log \pi_j + \sum_{i=1}^{|V|} log(1 + f_i) \log(Pr(i|j))$$

Also, in order to take stop words into account, we will add a Inverse Document Frequency (IDF)weight on each word:

$$t_i = \log(\frac{\sum_{n=1}^{N} doc_n}{doc_i})$$

$$Pr(j) = \log \pi_j + \sum_{i=1}^{|V|} f_i \log(t_i Pr(i|j))$$

Even though the stop words have already been set to 0 for this specific use case, t
implementation is being added to generalize the function.

```
#Calculate IDF
tot = len(df['docIdx'].unique())
pb_ij = df.groupby(['wordIdx'])
IDF = np.log(tot/pb_ij['docIdx'].count())
IDF_dict = IDF.to_dict()

def MNB(df, smooth = False, IDF = False):
    '''
    Multinomial Naive Bayes classifier
    :param df [Pandas Dataframe]: Dataframe of data
    :param smooth [bool]: Apply Smoothing if True
    :param IDF [bool]: Apply Inverse Document Frequency if True
    :return predict [list]: Predicted class ID
    '''
    #Using dictionaries for greater speed
```

```
        df_dict = df.to_dict()
        new_dict = {}
        prediction = []

        #new_dict = {docIdx : {wordIdx: count},....}
        for idx in range(len(df_dict['docIdx'])):
            docIdx = df_dict['docIdx'][idx]
            wordIdx = df_dict['wordIdx'][idx]
            count = df_dict['count'][idx]
            try:
                new_dict[docIdx][wordIdx] = count
            except:
                new_dict[df_dict['docIdx'][idx]] = {}
                new_dict[docIdx][wordIdx] = count

        #Calculating the scores for each doc
        for docIdx in range(1, len(new_dict)+1):
            score_dict = {}
            #Creating a probability row for each class
            for classIdx in range(1,21):
                score_dict[classIdx] = 1
                #For each word:
                for wordIdx in new_dict[docIdx]:
                    #Check for frequency smoothing
                    #log(1+f)*log(Pr(i|j))
                    if smooth:
                        try:
                            probability=Pr_dict[wordIdx][classIdx]
                            power = np.log(1+ new_dict[docIdx][wordIdx])
                            #Check for IDF
                            if IDF:
                                score_dict[classIdx]+=(
                                   power*np.log(
                                   probability*IDF_dict[wordIdx]))
                            else:
                                score_dict[classIdx]+=power*np.log(
                                                        probability)
                        except:
                            #Missing V will have log(1+0)*log(a/16689)=0
                            score_dict[classIdx] += 0
                    #f*log(Pr(i|j))
                    else:
                        try:
                            probability = Pr_dict[wordIdx][classIdx]
                            power = new_dict[docIdx][wordIdx]
                            score_dict[classIdx]+=power*np.log(
                                                  probability)
                            #Check for IDF
                            if IDF:
                                score_dict[classIdx]+= power*np.log(
                                       probability*IDF_dict[wordIdx])
                        except:
                            #Missing V will have 0*log(a/16689) = 0
                            score_dict[classIdx] += 0
```

```
            #Multiply final with pi
            score_dict[classIdx] +=  np.log(pi[classIdx])

        #Get class with max probabilty for the given docIdx
        max_score = max(score_dict, key=score_dict.get)
        prediction.append(max_score)

    return prediction
```

Comparing the effects of smoothing and IDF:

```
regular_predict = MNB(df, smooth=False, IDF=False)
smooth_predict  = MNB(df, smooth=True, IDF=False)
tfidf_predict   = MNB(df, smooth=False, IDF=True)
all_predict     = MNB(df, smooth=True, IDF=True)

#Get list of labels
train_label = pd.read_csv('20news-bydate/matlab/train.label',
                          names=['t'])
train_label= train_label['t'].tolist()

total = len(train_label)
models = [regular_predict, smooth_predict,
          tfidf_predict, all_predict]
strings = ['Regular', 'Smooth', 'IDF', 'Both']

for m,s in zip(models,strings):
    val = 0
    for i,j in zip(m, train_label):
        if i != j:
            val +=1
        else:
            pass
    print(s,"Error:\t\t",val/total * 100, "%")
```

```
Regular Error:          1.677167450527997 %
Smooth Error:           1.2867157689235957 %
IDF Error:              1.694915254237288 %
Both Error:             1.2867157689235957 %
```

As we can see, IDF has little effect as we removed the stop words. Smoothing, however, makes the model more accurate.

Hence, our optimal model is:

$$Pr(j) = \log \pi_j + \sum_{i=1}^{|V|} log(1 + f_i) \log(Pr(i|j))$$

## *Test Data*

Now that we have out model, let's use it to predict our test data.

```
#Get test data
test_data = open('20news-bydate/matlab/test.data')
df = pd.read_csv(test_data, delimiter=' ', names=['docIdx',
'wordIdx', 'count'])

#Get list of labels
test_label = pd.read_csv('/home/sadat/Downloads/HW2_210/20news-
bydate/matlab/test.label', names=['t'])
test_label= test_label['t'].tolist()

#MNB Calculation
predict = MNB(df, smooth = True, IDF = False)

total = len(test_label)
val = 0
for i,j in zip(predict, test_label):
    if i == j:
        val +=1
    else:
        pass

print("Error:\t",(1-(val/total)) * 100, "%")
```

## 1.9.3. Bernoulli Naive Bayes

`BernoulliNB` implements the naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions; i.e., there may be multiple features but each one is assumed to be a binary-valued (Bernoulli, boolean) variable. Therefore, this class requires samples to be represented as binary-valued feature vectors; if handed any other kind of data, a `BernoulliNB` instance may binarize its input (depending on the `binarize` parameter).

The decision rule for Bernoulli naive Bayes is based on

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i)$$

which differs from multinomial NB's rule in that it explicitly penalizes the non-occurrence of a feature $i$ that is an indicator for class $y$, where the multinomial variant would simply ignore a non-occurring feature.

In the case of text classification, word occurrence vectors (rather than word count vectors) may be used to train and use this classifier. `BernoulliNB` might perform better on some datasets, especially those with shorter documents. It is advisable to evaluate both models, if time permits.