Sentiment Analysis Using Naive Bayes

Ryan Slipe

Grand Canyon University

DSC-540-O500

8/23/2023

For this assignment we would like to know if we can use the Naive Bayes theorem to be an accurate classifier on a data set of movie reviews. This dataset was taken from the Kaggle dataset named "Bag of Words Meets Bag of Popcorn". The data consists of 50,000 IMBD movie reviews that do not include any personal information such as the persons' username or adress, or any other identifier that may cause harm. By understanding these are just personal opinions of people that we cannot identify, we deem this data set ethically safe to proceed.

Our main goal will be to answer the question, "Can we accurately predict the sentiment of a movie review using Naive Bayes?". We will also try to answer a second question of "Are we able to find words that most commonly belong to each sentiment?". The second question will be answered using the TfidfTransformer that will set weights to each word based on the importance of that word in the corresponding sentiment. We should expect to find positive words in the positive sentiments.

In [1]:
```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
```

In [2]:
```
# create data frame
df = pd.read_csv("C:/Users/ryans/Documents/Machine Learning GCU/Sentiment Analysis/bag_of_words.csv", delimiter = ',')
```

In [3]:
```
# view first 5 rows
df.head()
```

Out[3]:

	id	sentiment	review
0	5814_8	1	With all this stuff going down at the moment w...
1	2381_9	1	\The Classic War of the Worlds\" by Timothy Hi...
2	7759_3	0	The film starts with a manager (Nicholas Bell)...
3	3630_4	0	It must be assumed that those who praised this...
4	9495_8	1	Superbly trashy and wondrously unpretentious 8...

In [4]:
```
# check for null
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25000 entries, 0 to 24999
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   id         25000 non-null  object
 1   sentiment  25000 non-null  int64
 2   review     25000 non-null  object
dtypes: int64(1), object(2)
memory usage: 586.1+ KB
```

In [5]:
```
# view the tail
df.tail()
```

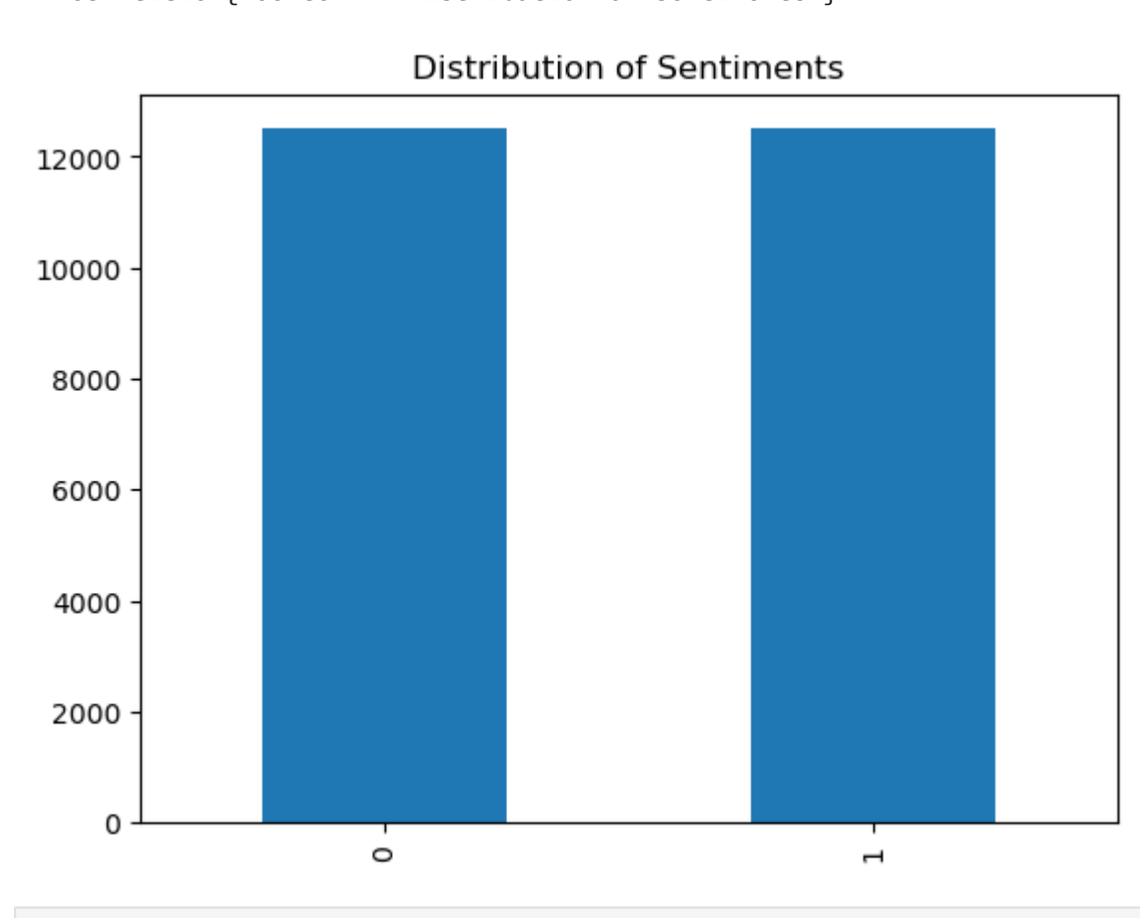
Out[5]:

	id	sentiment	review
24995	3453_3	0	It seems like more consideration has gone into...
24996	5064_1	0	I don't believe they made this film. Completel...
24997	10905_3	0	Guy is a loser. Can't get girls, needs to buil...
24998	10194_3	0	This 30 minute documentary Buñuel made in the ...
24999	8478_8	1	I saw this movie as a child and it broke my he...

We can see that our data is perfectly balanced so we will not have to worry about balancing the data later. This is important because it may make the accuracy of an unbalanced data set seem high when in reality it is not performing very well if one class is much more likely than the other.

In [6]:
```
df['sentiment'].value_counts().sort_index().plot(kind = 'bar', title = 'Distribution of Sentiments')
```

Out[6]:
```
<Axes: title={'center': 'Distribution of Sentiments'}>
```



In [7]:
```
# view value counts for sentiment
df['sentiment'].value_counts()
```

Out[7]:
```
1    12500
0    12500
Name: sentiment, dtype: int64
```

In [8]:
```
# view 10 positive sentiments
df[df['sentiment'] == 1][:10]
```

Out[8]:

	id	sentiment	review
0	5814_8	1	With all this stuff going down at the moment w...
1	2381_9	1	\The Classic War of the Worlds\" by Timothy Hi...
4	9495_8	1	Superbly trashy and wondrously unpretentious 8...
5	8196_8	1	I dont know why people think this is such a ba...
9	8713_10	1	 This movie is full of references. ...
11	6811_10	1	Although I generally do not like remakes belie...
12	11744_9	1	\Mr. Harvey Lights a Candle\" is anchored by a...
18	3304_10	1	This movie is one among the very few Indian mo...
19	9352_10	1	Most people, especially young people, may not ...
20	3374_7	1	\Soylent Green\" is one of the best and most d...

Before moving any further we see '< br />' is present in the 9th index. This should be removed and we can use str.replace to replace this string with a space instead. Any other punctuation can be removed using a RegexpTokenizer later, but we can use the string replace method to keep the string pattern simpler in our RegexpTokenizer.

In [58]:
```
df.iloc[0]['review']
```

Out[58]:
```
"With all this stuff going down at the moment with MJ i've started listening to his music, watching the odd documentary here and there, watched The Wiz and watched Moonwalke
r again. Maybe i just want to get a certain insight into this guy who i thought was really cool in the eighties just to maybe make up my mind whether he is guilty or innocen
t. Moonwalker is part biography, part feature film which i remember going to see at the cinema when it was originally released. Some of it has subtle messages about MJ's fee
ling towards the press and also the obvious message of drugs are bad m'kay.  Visually impressive but of course this is all about Michael Jackson so unless you remotely like
MJ in anyway then you are going to hate this and find it boring. Some may call MJ an egotist for consenting to the making of this movie BUT MJ and most of his fans would say
that he made it for the fans which if true is really nice of him.  The actual feature film bit when it finally starts is only on for 20 minutes or so excluding the Smooth Cr
iminal sequence and Joe Pesci is convincing as a psychopathic all powerful drug lord. Why he wants MJ dead so bad is beyond me. Because MJ overheard his plans? Nah, Joe Pesc
i's character ranted that he wanted people to know it is he who is supplying drugs etc so i dunno, maybe he just hates MJ's music.  Lots of cool things in this like MJ turni
ng into a car and a robot and the whole Speed Demon sequence. Also, the director must have had the patience of a saint when it came to filming the kiddy Bad sequence as usua
lly directors hate working with one kid let alone a whole bunch of them performing a complex dance scene.  Bottom line, this movie is for people who like MJ on one level or
another (which i think is most people). If not, then stay away. It does try and give off a wholesome message and ironically MJ's bestest buddy in this movie is a girl! Micha
el Jackson is truly one of the most talented people ever to grace this planet but is he guilty? Well, with all the attention i've gave this subject....hmmm well i don't know
because people can be different behind closed doors, i know this for a fact. He is either an extremely nice but stupid guy or one of the most sickest liars. I hope he is not
the latter."
```

In [13]:
```
# clean the text by replacing <br /> with a space
df['review'] = df['review'].str.replace('<br />', ' ')
```

In [14]:
```
# check that they have been removed
df.iloc[0]['review']
```

Out[14]:
```
"With all this stuff going down at the moment with MJ i've started listening to his music, watching the odd documentary here and there, watched The Wiz and watched Moonwalke
r again. Maybe i just want to get a certain insight into this guy who i thought was really cool in the eighties just to maybe make up my mind whether he is guilty or innocen
t. Moonwalker is part biography, part feature film which i remember going to see at the cinema when it was originally released. Some of it has subtle messages about MJ's fee
ling towards the press and also the obvious message of drugs are bad m'kay.  Visually impressive but of course this is all about Michael Jackson so unless you remotely like
MJ in anyway then you are going to hate this and find it boring. Some may call MJ an egotist for consenting to the making of this movie BUT MJ and most of his fans would say
that he made it for the fans which if true is really nice of him.  The actual feature film bit when it finally starts is only on for 20 minutes or so excluding the Smooth Cr
iminal sequence and Joe Pesci is convincing as a psychopathic all powerful drug lord. Why he wants MJ dead so bad is beyond me. Because MJ overheard his plans? Nah, Joe Pesc
i's character ranted that he wanted people to know it is he who is supplying drugs etc so i dunno, maybe he just hates MJ's music.  Lots of cool things in this like MJ turni
ng into a car and a robot and the whole Speed Demon sequence. Also, the director must have had the patience of a saint when it came to filming the kiddy Bad sequence as usua
lly directors hate working with one kid let alone a whole bunch of them performing a complex dance scene.  Bottom line, this movie is for people who like MJ on one level or
another (which i think is most people). If not, then stay away. It does try and give off a wholesome message and ironically MJ's bestest buddy in this movie is a girl! Micha
el Jackson is truly one of the most talented people ever to grace this planet but is he guilty? Well, with all the attention i've gave this subject....hmmm well i don't know
because people can be different behind closed doors, i know this for a fact. He is either an extremely nice but stupid guy or one of the most sickest liars. I hope he is not
the latter."
```

In [15]:
```
df.shape
```

Out[15]:
```
(25000, 3)
```

Preprocessing Using Examples

In [16]:
```
# libraries needed for our sentiment analysis
import nltk
from nltk import word_tokenize
```

The first thing we will have to do for our data is 'tokenize' the review. This means we will have to separate the string into separate words. We can do this using the word_tokenize function provided in nltk library. As we see from our example, every word is taken out of the string as its own entity and the punctuation also is included.

In [17]:
```
# example sentence to tokenize
example = 'This is an example of word tokenize.'

# print the results
print(word_tokenize(example))
```

```
['This', 'is', 'an', 'example', 'of', 'word', 'tokenize', '.']
```

Since we will be using these sentence reviews to classify them as being positive or negative, we can remove words that will appear in both without describing the actual sentence. These are called 'stop words' and we have a few in our example sentence. We can import stop_words from nltk. After importing stopwords we can print the stop words to get an idea of what they are. We can also use the stopwords on our example to see how it is changed. This will clean our data. By looking at some of our reviews we can also see punctuation such as "/" that will have to be cleaned as well.

After removing the stop words we should also remove our punctuation since word_tokenize() will separate them like any other word but include them in the tokenized version. This is why we should remove the punctuation before using word_tokenize(). Also, we should do this before using stopwords because they will be working with the tokenized string. In our data we saw there was a \ so after cleaning our data we should make sure to double check those rows to ensure it was removed with the punctuation. If it was not we can set up our own custom stop words.

In [18]:
```
# import stop words
from nltk.corpus import stopwords
```

In [19]:
```
# set a variable equal to the english stopwords
stop_words = set(stopwords.words('english'))

# print all english stop words
print(stop_words)
```

```
{"doesn't", 'needn', 'as', 'at', 've', 'some', "isn't", 'mightn', "mustn't", 'nor', 'won', 'of', 'in', 'did', 'above', 'that', 'each', 'had', 'were', 're', 'no', 'itself',
'haven', 'because', 'has', 'our', "it's", 'who', 'after', 'into', 'again', 'same', 'shouldn', 'didn', 'y', 'over', 'both', 'or', "weren't", 'about', "you'll", 'himself', 'it
s', 'your', 'during', 'when', 'this', "haven't", 'hadn', 'here', 'having', 'there', 'more', 'them', 'm', 'up', 'she', 'hasn', 'they', 'a', 'i', 'than', 'ourselves', 'doing',
'mustn', 'her', 'those', 'was', 'don', 'theirs', "you've", 'will', "aren't", 'o', 'between', 'off', 'own', 'ain', 'weren', 'under', "should've", 'are', 'for', 'where', "has
n't", 'by', 't', 'these', "you'd", 'him', "mightn't", 'down', 'below', 'against', 'doesn', "you're", 'too', 'you', 'to', 'the', 'which', 'before', 'my', 'aren', 'yourselve
s', 'with', 'ours', 's', 'whom', "hadn't", "wouldn't", 'through', 'wouldn', 'what', 'then', 'further', 'ma', 'herself', 'being', 'on', 'so', 'most', 'wasn', "couldn't", 'the
mselves', "shouldn't", 'be', 'once', 'should', 'he', 'other', 'all', 'couldn', 'few', 'we', 'll', 'am', "she's", 'now', 'his', "wasn't", 'hers', 'out', 'me', 'yours', 'thei
r', "won't", 'does', 'such', 'an', 'if', 'while', 'from', "shan't", "that'll", 'it', 'myself', "didn't", 'only', 'isn', 'shan', 'but', 'is', 'been', 'very', 'and', "don't",
'd', 'not', 'why', 'just', 'can', 'have', 'how', 'do', 'until', 'yourself', 'any', "needn't"}
```

In [20]:
```
# import string to help us remove the punctuation and join letters to form words
import string
```

In [21]:
```
# punctuation removal example
punc_example = 'Next, let"s remove the punctuation!'

# for loop to remove punctuation
punc_example1 = [char for char in punc_example if char not in string.punctuation]

# results
print(punc_example1)
```

```
['N', 'e', 'x', 't', ' ', 'l', 'e', 't', 's', ' ', 'r', 'e', 'm', 'o', 'v', 'e', ' ', 't', 'h', 'e', ' ', 'p', 'u', 'n', 'c', 't', 'u', 'a', 't', 'i', 'o', 'n']
```

In [22]:
```
# join the words together based on the whitespace created from removing punctuation
punc_example1 = ''.join(punc_example1)

# print results
print(punc_example1)
```

```
Next lets remove the punctuation
```

In [23]:
```
# use word_tokenize now then use stopwords
tokenized = word_tokenize(punc_example1)
tokenized
```

Out[23]:
```
['Next', 'lets', 'remove', 'the', 'punctuation']
```

In [24]:
```
# remove the stop words with a function similar to our punctuation one. ('the')
filtered_sentence = [word for word in tokenized if not word.lower() in stop_words]

# print the results
print(filtered_sentence)
```

```
['Next', 'lets', 'remove', 'punctuation']
```

At this point our text data is cleaned but is not ready for any machine learning analysis. Our machine learning algorithm will have to work with numerical numbers so we will have to create a document term matrix (DTM). This will assign each word to an integer in a matrix throughout our data. We can do this using CountVectorizer from sklearn.feature_extraction.text. We can demonstrate how this works on a single sentence before using it on all of our data.

In [25]:
```
# We can use CountVectorizer so that we are working with numbers
from sklearn.feature_extraction.text import CountVectorizer
```

Like many other modules in sklearn we will create an object then fit our data to the object then transform our data using the object. After fitting our test example to the CountVectorizer object we can use .vocabulary_ to see what is happening. This has created a dictionary that stores each word as an integer. Our test sentiment has the word test listed two times so the dictionary only has test listed once. In our example, test is assigned to the value 2. We can also see that the stopwords have not been removed, but the word 'a' is gone because CountVectorizer counts words that are atleast two characters in length.

In [26]:
```
# sentiment example
test = ["This is a test that's called test."]

# initiate a CountVectorizer object
vectorizer = CountVectorizer()

# fit the vectorizer on our data
vectorizer.fit(test)

# print the fitted data (not transformed yet)
print(vectorizer.vocabulary_)
```

```
{'this': 4, 'is': 1, 'test': 2, 'that': 3, 'called': 0}
```

Once the data is transformed using our vectorizer we can see that a matrix is created. It lists the row as 0 and the dictionary values for each corresponding word. Our vectorizer had 5 words and we see that in the first row, which is the entire sentence, the value (0, 2) corresponds to a 2 while all others correspond to a 1. This is the case because the word 'test' appears in the sentence twice. This will be important information for us when we want to figure out which words are more important than others in our sentiments.

In [27]:
```
# transform our data
vector = vectorizer.transform(test)
print(vector)
```

```
  (0, 0)	1
  (0, 1)	1
  (0, 2)	2
  (0, 3)	1
  (0, 4)	1
```

Preprocessing Our Data

Now that we have run through some basic examples on what we have to do to the sentiments in our data we can use these techniques to preprocess our data. We went through numerous steps that included:

- removing punctuation
- joining the words back together
- word_tokenize()
- remove stopwords

We can create a function that lists these steps and pass our dataframe into it.

In [28]:
```
def text_preprocessing(review):

    # remove punctuation
    punctuation_removal = [char for char in review if char not in string.punctuation]

    # join the characters back into words based on whitespace
    punctuation_removal = ''.join(punctuation_removal)

    # tokenize our sentiments and return them using .lower so they are all formatted equally
    return [word for word in word_tokenize(punctuation_removal) if word.lower() not in stopwords.words('english')]
```

In [29]:
```
# this cell has been commented out because of long run-time

# create new column of processed sentiment in our data frame by apply and our function
# df['preproccesed'] = df.iloc[:, 2].apply(text_preprocessing)
```

After creating the function we can pass our column through using .apply() but since we are working with a large dataset it will take a long time making it computationally expensive. Luckily for us, we can use RegexpTokenizer from nltk.tokenize to speed up this process. When using CountVectorizer there are parameters for both stop words and tokenizer. We can pass through the english stopwords and create a tokenizer from nltk that will tokenize our words based on the pattern. When using the regexptokenizer we will state that we want our pattern to be all lower case and upper case letters and all numbers. This will keep the puncuation out when tokenizing which is what we did when we made the punctuation_removal for loop.

In [30]:
```
from nltk.tokenize import RegexpTokenizer
```

This can streamline our preprocessing into just a few steps and run much faster than our apply method. Below we have a sample string that we will remove punctuation, stop words, tokenize, and transform into a document term matrix. First, we set up a RegexpTokenizer object with a pattern that we want to look for. This will remove the punctuation from our string. Then we set up our DTM by using a CountVectorizer and remove stopwords by stating 'english' in that parameter. Then, we use the tokenizer that we established as another parameter. The resulting matrix shows there are no stop words and that machine and learning are counted twice as they should be.

In [31]:
```
# create a sample string
string = ['I love machine learning, Machine Learning 101 is fun.']

# set up a tokenizer looking for the pattern of letters or numbers
toke = RegexpTokenizer('[A-Za-z0-9]+')

# create a CountVectorizer using the parameters for stop words and our tokenizer
vec = CountVectorizer(stop_words='english', tokenizer= toke.tokenize)

# fit the CountVectorizer
dtm = vec.fit(string)

# view the digits dictionary for the matrix
print(dtm.vocabulary_)
```

```
{'love': 3, 'machine': 4, 'learning': 2, '101': 0, 'fun': 1}
C:\Users\ryans\anaconda_3\lib\site-packages\sklearn\feature_extraction\text.py:528: UserWarning: The parameter 'token_pattern' will not be used since 'tokenizer' is not Non
e'
  warnings.warn(
```

In [32]:
```
# create a variable for our matrix
matrix = dtm.transform(string)

# print the matrix to make sure machine and learning are there twice.
print(matrix)
```

```
  (0, 0)	1
  (0, 1)	1
  (0, 2)	2
  (0, 3)	1
  (0, 4)	2
```

Now we will run a test to see how we tokenize data that is in our dataframe. We select the second row of our review column and tokenize that sentiment. This has removed any punctuation and next we would pass this tokenizer into our CountVectorizer.

In [33]:
```
# run a test on the second row (1 index)
data_Test = df.iloc[1]['review']
```

In [34]:
```
# create a tokenizer with all letters and numbers
data_toke = RegexpTokenizer('[A-Za-z0-9]+')
tokens = data_toke.tokenize(data_Test)
print(tokens)
```

```
['The', 'Classic', 'War', 'of', 'the', 'Worlds', 'by', 'Timothy', 'Hines', 'is', 'a', 'very', 'entertaining', 'film', 'that', 'obviously', 'goes', 'to', 'great', 'effort',
'and', 'lengths', 'to', 'faithfully', 'recreate', 'H', 'G', 'Wells', 'classic', 'book', 'Mr', 'Hines', 'succeeds', 'in', 'doing', 'so', 'I', 'and', 'those', 'who', 'watche
d', 'his', 'film', 'with', 'me', 'appreciated', 'the', 'fact', 'that', 'it', 'was', 'not', 'the', 'standard', 'predictable', 'Hollywood', 'fare', 'that', 'comes', 'out', 'ev
ery', 'year', 'e', 'g', 'the', 'Spielberg', 'version', 'with', 'Tom', 'Cruise', 'that', 'had', 'only', 'the', 'slightest', 'resemblance', 'to', 'the', 'book', 'Obviously',
'everyone', 'looks', 'for', 'different', 'things', 'in', 'a', 'movie', 'Those', 'who', 'envision', 'themselves', 'as', 'amateur', 'critics', 'look', 'only', 'to', 'criticiz
e', 'everything', 'they', 'can', 'Others', 'rate', 'a', 'movie', 'on', 'more', 'important', 'bases', 'like', 'being', 'entertained', 'which', 'is', 'why', 'most', 'people',
'never', 'agree', 'with', 'the', 'critics', 'We', 'enjoyed', 'the', 'effort', 'Mr', 'Hines', 'put', 'into', 'being', 'faithful', 'to', 'H', 'G', 'Wells', 'classic', 'novel',
'and', 'we', 'found', 'it', 'to', 'be', 'very', 'entertaining', 'This', 'made', 'it', 'easy', 'to', 'overlook', 'what', 'the', 'critics', 'perceive', 'to', 'be', 'its', 'sho
rtcomings']
```

After running these tests and examples, we know what is happening to the data. After clarifying the results we are ready to implement this preprocessing on our dataframe. Printing the vocabulary dictionary for our dataframe will not give us much useable information because there are too many words for us to gain any insights from, but we can print it to show that the process is the same.

In [91]:
```
# establish tokenizer we will be using
tokenizer = RegexpTokenizer('[A-Za-z0-9]+')

# create CountVectorizer using the stopwords and tokenizer
count_vect = CountVectorizer(stop_words = 'english', tokenizer = tokenizer.tokenize)

# fit our data to the CountVectorizer
our_count_vect = count_vect.fit(df['review'])

# view (very long) vocabulary dictionary
# print(our_count_vect.vocabulary_)
```

```
C:\Users\ryans\anaconda_3\lib\site-packages\sklearn\feature_extraction\text.py:528: UserWarning: The parameter 'token_pattern' will not be used since 'tokenizer' is not Non
e'
  warnings.warn(
```

In [36]:
```
# transform our data
dtm = our_count_vect.transform(df['review'])
```

We can print the first sentence to see that it is very long and has the row number 0 along with the word that has been transformed into an integer. Many words have only been used one time.

In [37]:
```
# print the first sentence
print(dtm[0:1])
```

```
  (0, 437)	1
  (0, 1566)	1
  (0, 4751)	1
  (0, 5115)	1
  (0, 5362)	3
  (0, 6886)	1
  (0, 7240)	1
  (0, 7321)	1
  (0, 8347)	1
  (0, 9324)	1
  (0, 9491)	1
  (0, 10090)	1
  (0, 10405)	1
  (0, 11192)	1
  (0, 11429)	1
  (0, 12265)	1
  (0, 12734)	1
  (0, 13583)	1
  (0, 14073)	1
  (0, 14453)	1
  (0, 14486)	2
  (0, 14943)	1
  (0, 15340)	1
  (0, 16194)	1
  (0, 16518)	1
  :	:
  (0, 63262)	1
  (0, 63359)	1
  (0, 63478)	1
  (0, 63965)	1
  (0, 64604)	1
  (0, 64779)	1
  (0, 65919)	1
  (0, 65921)	1
  (0, 66012)	1
  (0, 67740)	1
  (0, 67757)	1
  (0, 67790)	1
  (0, 68004)	1
  (0, 69213)	1
  (0, 69911)	1
  (0, 70294)	2
  (0, 70953)	1
  (0, 71455)	1
  (0, 71458)	1
  (0, 71463)	1
  (0, 71618)	2
  (0, 71624)	1
  (0, 72258)	1
  (0, 72727)	1
  (0, 72963)	1
```

Building a Model

For this assignment, we will be using the Naive Bayes classifier algorithm as our model. In this model we will be specifically using the multinominal Naive Bayes algorithm to make predictions on the sentiments. We choose this type of Naive Bayes over others such as the Gaussian approach because the multinomial technique is known to work best with categorical or discrete variables whereas the Gaussian approach is used for continuous variables. This classifier works with probabilities and treats all variables as being independent. The variables in our data are the different words and they will be treated as being independent. This means the probability of a single word is not dependent on the occurence of any other word. As we have seen previously each word in our data will be given an integer as its ID. This means every sentence will be a collection of integers followed by the final class of the sentence being 1 for positive and 0 for negative. A simple illustration of a sentence consisting of the string "This movie is good" can be integers such as 0, 1, 2, 3 and the sentiment will be 1 for positive. These integers are then used in the formula to find the corresponding probabilities along with the final sentiment class. With over 10,000 reviews, we may find that the word "good" has a high probability of being classified as a positive sentiment. This is found using conditional probabilities. The Bayes' Theorem is as follows:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

The | symbol in the equation means "given" so we are solving the Probability(A given B has already occured). In our example we would multiply the probability that the sentiment is a 1 times the probability of each word in the sentence given the sentiment is a 1. The probability of B | A is the probability of the word given the sentiment. In a text classifyer the P(B) will be each of our features which are the independent words so we will have $x_1, x_2, ... x_n$ for each word in our phrases

This means x_0 can be a word such as "good" and we will find the probability of "good" given the probability that the sentiment is a 1. This probability is used on every sentence in our data, not just the sentence that we are looking at so we would look through all the sentences for the probability of "good" appearing | the sentiment was positive. As you can tell, this is a lot of arithmetic even though it is simple arithmetic, errors are likely to occur and many words are likely to be in a review, so the sklearn MultinominalNB comes in handy.

The first step in model building will be to train our model. We will use a train test split to train our model and then test that model on the test set. The X_train will consist of 80% of our reviews that have been transformed into a document term matrix and the y_train will be the sentiment. The model will learn from the document term matrix about what classifies it as being either a good or bad sentiment.

In [38]:
```
# import library
from sklearn.model_selection import train_test_split
```

In [39]:
```
# split our data for training and testing
X_train, X_test, y_train, y_test = train_test_split(dtm, df['sentiment'], test_size = 0.2, random_state = 0)
```

In [40]:
```
# import the Naive Bayes algorithm
from sklearn.naive_bayes import MultinomialNB
```

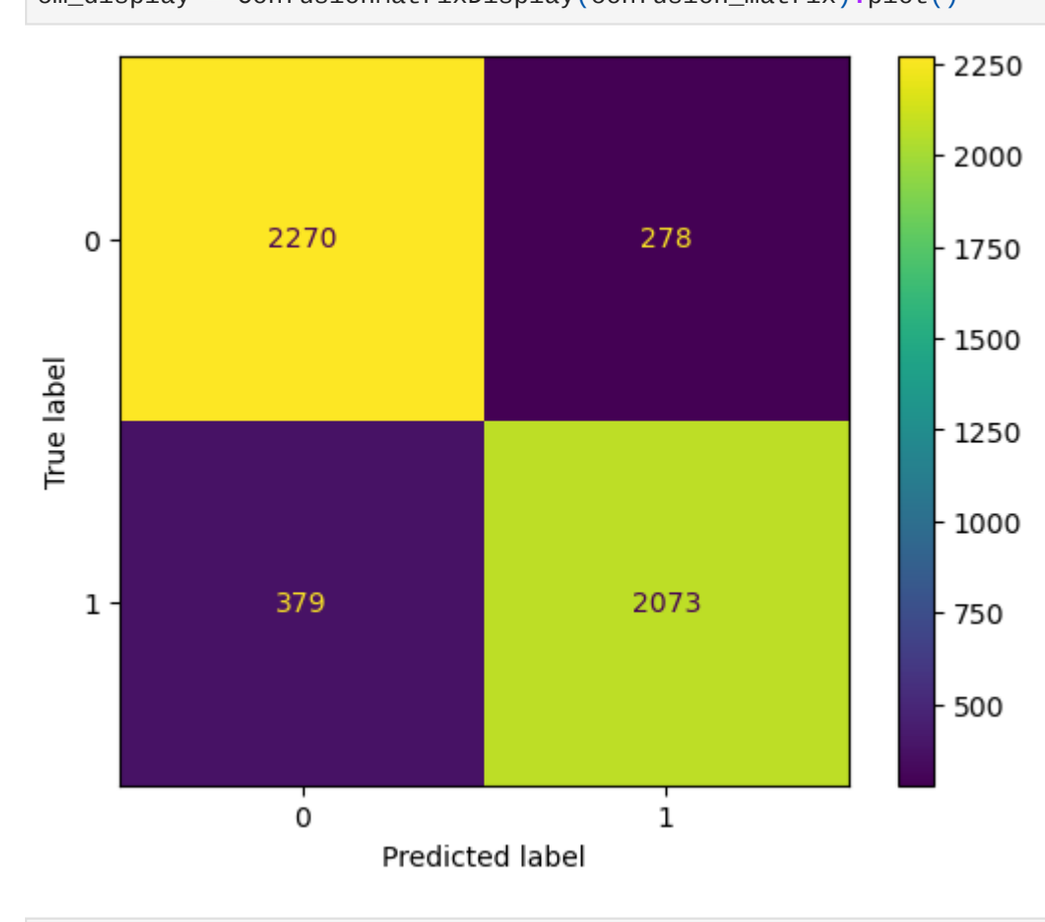
In [41]:
```
# create a multinomial Naive Bayes object
MNB = MultinomialNB()

# fit to our data
model = MNB.fit(X_train, y_train)
```

In [42]:
```
# Evaluation and Predictions
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay, RocCurveDisplay, roc_curve, roc_auc_score
from sklearn.metrics import confusion_matrix
```

In [44]:
```
# make predictions
predicted = MNB.predict(X_test)

# generate accuracy score
accuracy_score = accuracy_score(predicted, y_test)

# print our score
print(accuracy_score * 100)
```

```
86.86
```

In [45]:
```
# confusion matrix
confusion_matrix = confusion_matrix(y_test, predicted)
cm_display = ConfusionMatrixDisplay(confusion_matrix).plot()
```

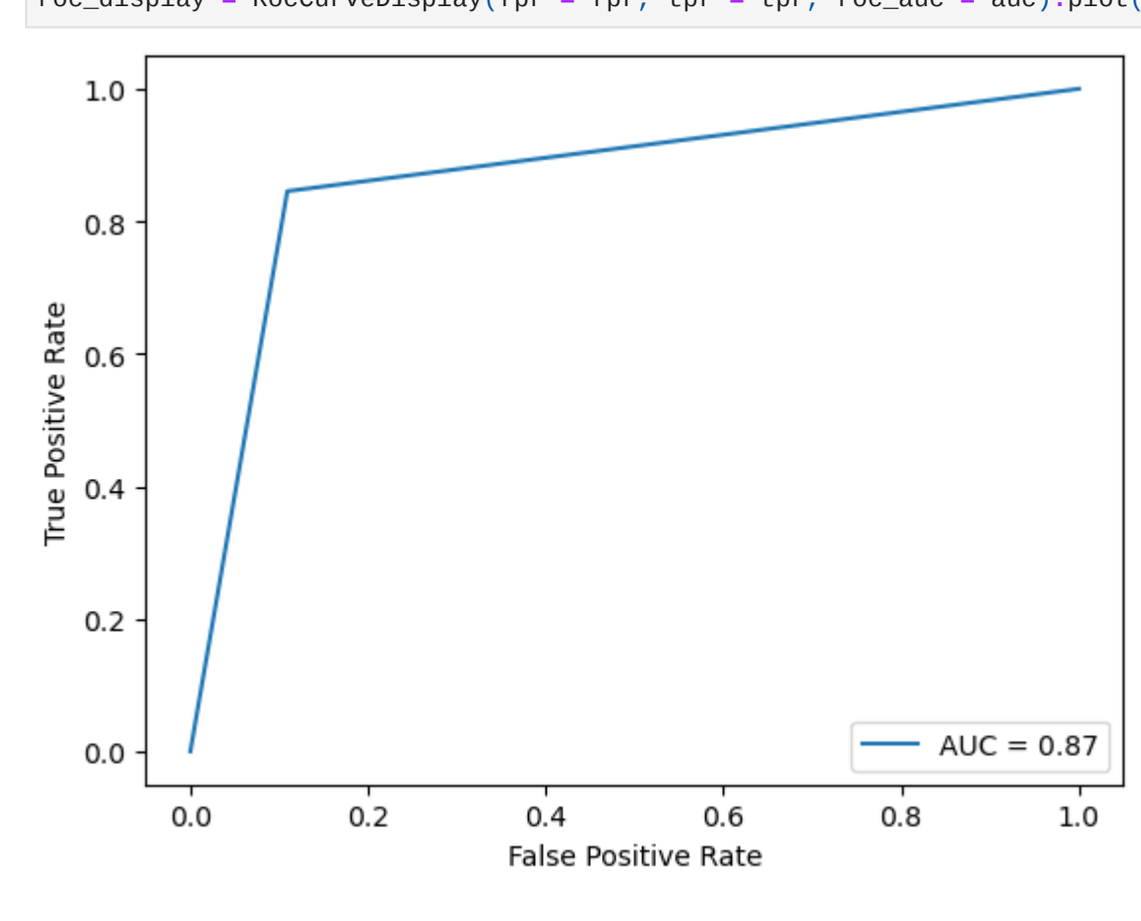


In [46]:
```
# classification report
from sklearn.metrics import classification_report

# show results
print(classification_report(y_test, predicted))
```

```
              precision    recall  f1-score   support

           0       0.86      0.89      0.87      2548
           1       0.88      0.85      0.86      2452

    accuracy                           0.87      5000
   macro avg       0.87      0.87      0.87      5000
weighted avg       0.87      0.87      0.87      5000
```

The ROC curve is a popular metric when working with binary classifiers like we are in this assignment. The ROC curve is a plot of the true positive rate against the false positive rate. This can also be known as plotting sensitivity vs. 1 - specificity. A random classifier would have a diagonal line through the middle of the graph and the firther we are towards the top left corner, the better the results.

In [47]:
```
# ROC Curve and AUC Score
fpr, tpr, _ = roc_curve(y_test, predicted)

# AUC
auc = roc_auc_score(y_test, predicted)

# display results
roc_display = RocCurveDisplay(fpr = fpr, tpr = tpr, roc_auc = auc).plot()
```



We can use our model to make predictions on reviews that have not been made before. To do this, we pass our model followed by .predict(). In the predict function we will need to pass through the review but make sure that we transform it into a dtm first so we will use our_count_vect() and pass the new review in to it. We begin by making a prediction on a sentence that should be received positively and follow it up with a sentence that should be classified as a negative sentence. As a reminder, 1 = positive and 0 = negative.

In [48]:
```
# make predictions of a new sentiment that should be positive
model.predict(our_count_vect.transform(['I love this movie']))
```

Out[48]:
```
array([1], dtype=int64)
```

In [49]:
```
# make prediction of a new sentiment that should be negative
model.predict(our_count_vect.transform(['I hate this movie']))
```

Out[49]:
```
array([0], dtype=int64)
```

Trying TFIDF

TfidfTransformer will transform our words and assign a weight to each word based on importance. The higher the value, the more important the word. For movie reviews we would assume that words like "movie" and "film" are present in many of the reviews whether they are good or bad sentiments. The tfidf assigns weights to words and words like "movie" and "film" would have a very low weight because they will not be adding much information as to what class they belong in when that word is present. In this sentence we see the word that was given the integer value of 14486 has high significance. We can see what this integer is in a word form. This word turns out to be 'cool' which we would think should be used highly in a positive sentiment so we can check if this was indeed a positive sentiment. (1 = positive)

In [50]:
```
# import the TfidfTransformer from sklearn.feature_extraction.text
from sklearn.feature_extraction.text import TfidfTransformer
```

Here we create a variable tfidf_transformer and create the transformer. In the same line we also fit and transform our data by passing through the variable dtm. The variable dtm is our CountVectorizer after we removed the stop words, tokenized the data and fit_transformed the data. The data that was transformed was the 'review' column of our data that we stored as a dataframe called df.

In [59]:
```
# fit and transform our data
tfidf_transformer = TfidfTransformer().fit_transform(dtm)

print(tfidf_transformer[0:1])
```

```
  (0, 72963)	0.036562409582333914
  (0, 72727)	0.07238098099410983
  (0, 72258)	0.059335077800039715
  (0, 71624)	0.023323500236584128
  (0, 71618)	0.05735460035755999
  (0, 71463)	0.03323606938973989
  (0, 71458)	0.0326595629792587
  (0, 71455)	0.02502438709579382
  (0, 70953)	0.045732329082445664
  (0, 70294)	0.04602339471361909
  (0, 69911)	0.03514261461934535
  (0, 69213)	0.0376153984327079
  (0, 68004)	0.04308702479632016
  (0, 67790)	0.03008004544850767
  (0, 67757)	0.030740249749913948
  (0, 67740)	0.028685508660906633
  (0, 66012)	0.025525774562618703
  (0, 65921)	0.02047440638387402
  (0, 65919)	0.025309019675307503
  (0, 64779)	0.03919273409794134
  (0, 64604)	0.012186589464599265
  (0, 63965)	0.07075436933878564
  (0, 63478)	0.04150465929565712
  (0, 63359)	0.038099094917003244
  (0, 63262)	0.03177140216553855
  :	:
  (0, 16518)	0.031108269914948573
  (0, 16194)	0.03973591456645362
  (0, 15340)	0.044487609207865486
  (0, 14943)	0.027858616567271264
  (0, 14486)	0.07139722026617694
  (0, 14453)	0.039800375147964195
  (0, 14073)	0.0756676306424658
  (0, 13583)	0.0418098098357827
  (0, 12734)	0.05380708249799628
  (0, 12265)	0.032753975241015955
  (0, 11429)	0.02141908199731113
  (0, 11192)	0.03707759712194649
  (0, 10405)	0.03505204523680799
  (0, 10090)	0.030724392147059257
  (0, 9491)	0.03673791577064316
  (0, 9324)	0.0473253759883313
  (0, 8347)	0.030862866447978542
  (0, 7321)	0.026568930793461436
  (0, 7240)	0.05610869516095996
  (0, 6886)	0.0812861953270466
  (0, 5362)	0.05944375827999956
  (0, 5115)	0.027044351293860083
  (0, 4751)	0.03564029423115286
  (0, 1566)	0.03700851253836679
  (0, 437)	0.03781555367384762
```

This brings us to our second question that we were trying to solve when starting this project. We would like to be able to extract the word that is of high importance to see if it makes sense from a common knowledge standpoint. We would like to see a positive word associated with a positive sentiment which will also be a way to check that our model is performing correctly. To find out which word belongs to the integer value given in the CountVectorizer/TfidfTransformer, we use the count_vect variable which is the CountVectorizer that removes stop words and tokenizes the words before we fit and transform our column of interest. We then pass .get_feature_names_out() followed by the number of interest to return its original word.

In [60]:
```
# print the value to an important key in the sentence
count_vect.get_feature_names_out()[14486]
```

Out[60]:
```
'cool'
```

To see what sentiment this sentence had we can use iloc['row number']['sentiment column'] to return the sentiment that contained this sentence.

In [61]:
```
# 'cool' sounds like a word that would indicate a positive sentiment. Check it.
df.iloc[0]['sentiment']
```

Out[61]:
```
1
```

Using Tfidf in Our Model

To use Tfidf in our model we will need to split the data again this time using the tfidf_transformer as the predictor and the sentiment from our dataframe as the target column. To fit the data we can use the MNB model that we created previously and pass through our new X_train, y_train.

In [65]:
```
# split the data using the Tfidf
X_train, x_test, y_train, y_test = train_test_split(tfidf_transformer, df['sentiment'], test_size = 0.2, random_state = 0)

# fit the model to new splits
model = MNB.fit(X_train, y_train)
```

In [66]:
```
# make predictions
predicted = model.predict(X_test)
```

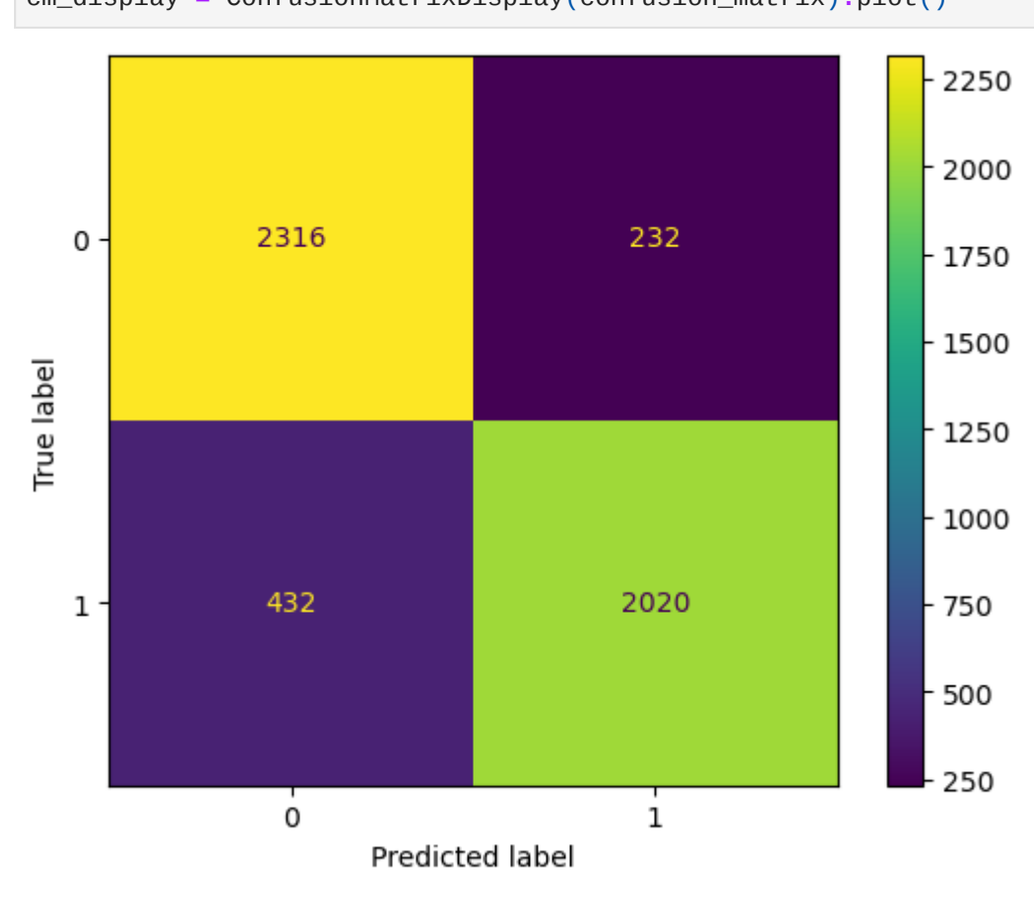
In [71]:
```
from sklearn import metrics
# accuracy score
score = metrics.accuracy_score(predicted, y_test)
```

This performs very similarly to our model, but slightly worse so we would use the first model as our final model. This model generates more true negatives, but is performing worse in true positives. This may be the case when slag or other natural language usage will throw off the original meaning of a word. When reviewing a superhero movie someone may enjoy the movie but review it by saying "The bad guy!". As humans we can see that they liked the bad guy, but bad is a word that is more frequently used with negative sentiments so this would be treated as a negative sentiment when it was actually a positive sentiment. In the confusion matrix, the numbers along the diagonal are the predictions that were made accurately by our model. Overall, both of these models are fairly accurate and can produce good results for us if we are to implement them.

In [73]:
```
# print our score
score * 100
```

Out[73]:
```
86.72
```

In [75]:
```
# confusion matrix
confusion_matrix = metrics.confusion_matrix(y_test, predicted)
cm_display = ConfusionMatrixDisplay(confusion_matrix).plot()
```



Cross- Validation

Before submitting our model for deployment we should cross validate it using kfolds. We will choose 10 folds to split up our data into training and testing. This ensures that the model is not learning specific information so that it will perform poorly when tested. The train test splits are done ten different times in this case and the results are fairly similar which means we can trust that our model will perform around the 86% accuracy rate that it got.

In [85]:
```
from sklearn.model_selection import KFold, cross_val_score
```

In [86]:
```
# cross-validation variable
cv = KFold(n_splits=10, random_state=0, shuffle=True)
```

In [89]:
```
scores = cross_val_score(model, dtm, df['sentiment'],
                         cv=cv, n_jobs=-1)
```

In [90]:
```
scores
```

Out[90]:
```
array([0.8668, 0.8728, 0.8604, 0.844 , 0.8576, 0.852 , 0.852 , 0.852 ,
       0.8552, 0.85  ])
```

References

Grus, J. (2015). Data Science From Scratch. O'Reilly Media, Inc.

Harrington, P. (2012). Machine Learning in Action. Manning Publications Co.

Patel, D. (2019). Machine Learning Tutorial Python 12 - K Fold Cross Validation[Video]. https://www.youtube.com/watch?v=gJo0uNL-5Qw&t=2s

Latrose, D., Latrose, C. (2015). Data Mining and Predictive Analysis. John Wiley & Sons

In [ ]: