# Assignment 5: Naive Bayes Classifer for Text Classifcation - Machine Learning 6316

Jacob Dineen / JD5ED

November 27, 2019

## 1 Naive Bayes Classifier for Movie Review Text Classification

This will be a fairly lengthy write up, and will be a bit redundant at times because I plan to essentially do the assignment twice: Once using the 'First Choice' preprocessing techniques, which involves a bit of manual parsing, and once using NLTK in which I employ some additional preprocessing techniques on the entire corpus (stopwords, tokenizing, lemmatization, etc..).

I want to make a couple of important notes on the structure of the py file that will be submitted.

- There is a fetch files method that is run through when the program launches. This will reference the first system argument and load all of the absolute file paths pertaining to the four folders into four separate lists. The parameter mainpath will point to the overarching datasets folder:

```
    def fetch_files(main_path):
        '''
        Params:
            None
        Returns:
            4 lists containing all absolute paths to train and test sets
                train_pos
                train_neg
                test_pos
                test_neg
        '''
        train_path = main + '\\training_set'
        test_path = main + '\\test_set'

        train_pos = train_path + '\\' + os.listdir(train_path)[1] + '\\'
        train_neg = train_path + '\\' + os.listdir(train_path)[0] + '\\'
        test_pos = test_path + '\\' + os.listdir(test_path)[1] + '\\'
        test_neg = test_path + '\\' + os.listdir(test_path)[0] + '\\'

        f = lambda x: [x + os.listdir(x)[i] for i in range(0,len(os.listdir(x)))]
        train_pos = f(train_pos)
        train_neg = f(train_neg)
        test_pos = f(test_pos)
        test_neg = f(test_neg)
        return train_pos, train_neg, test_pos, test_neg

```

Listing 1: Fetching Paths

- I made a comment on Piazza about the sys.argv[2] being unnecessary, as the code that point to it is commented out. Try-catching will be implemented in main to keep the same arg structure as the template notes.

### 1.1: Preprocessing

For 'First Choice' Preprocessing, the heuristics are as follows - Note that I built this into my transfer method, which is ultimately called by loadData. We read in each file from the specified directory (there are 4, and they need to stay somewhat isolated from eachother for counting purposes / MLE). First, we store the file in a temp variable. The way numpy reads this in is as an array of lists for each file. We instantiate a zero vector with cardinality = $|Vocab|$. We loop through the vocabulary set, and through the lists within the document array. If any form of 'love' is present in the list, we replace it with 'love' to add to our counts - This will be automatically done with NLTK stemming and lemmatization. Then we tokenize the file. If any word in the vocab is present within the tokenized representation of the document, we increment the bag of words vector. This ultimately needs to be called for every file within every directory. *Everything not binned into a specific word will increment the UNK token count.

```
def transfer(fileDj, vocabulary, love_replace = True):
    file = np.loadtxt(fileDj, delimiter='\n', dtype=str)
    BOWDj = [0] * len(vocabulary)
    for i,p in enumerate(vocabulary):
        for j in file:
            if love_replace:
                j = j.replace("loved", "love")
                j = j.replace("loves", "love")
                j = j.replace("loving", "love")
            tokens = j.split()
            for k in tokens:
                if p == k:
                    BOWDj[i] += 1
                else:
                    BOWDj[-1] += 1

    return BOWDj

```

Listing 2: Preprocessing

```
array(["films adapted from comic books have had plenty of success , wh
ether they're about superheroes ( batman , superman , spawn ) , or gea
red toward kids ( casper ) or the arthouse crowd ( ghost world ) , but
there's never really been a comic book like from hell before . ",
       "for starters , it was created by alan moore ( and eddie campbe
ll ) , who brought the medium to a whole new level in the mid '80s wit
h a 12-part series called the watchmen . ",
       'to say moore and campbell thoroughly researched the subject of
jack the ripper would be like saying michael jackson is starting to lo
ok a little odd . ',
       'the book ( or " graphic novel , " if you will ) is over 500 pa
ges long and includes nearly 30 more that consist of nothing but footn
otes . ',
       "in other words , don't dismiss this film because of its source
. ",
```

```
[1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 3, 0]
```

Figure 1: Fig 1 shows an example of the BoW transformation for the first positive train example.

$vocab = \{'love', 'wonderful', 'best', 'great', 'superb', 'still', 'beautiful', 'bad', 'worst', 'stupid', 'waste', 'boring', '?', '!'\}$

We can look at the Vocab set to understand the frequency of word occurrence: here, love, great and bad appear once and a ? appears three times. Now, we need to repeat this process for every document in our train set while keeping the documents of opposing polarity distinct.
****The vocab list was updated, and now we are reading in the 100 word vocabularly from the provided text file. I'm leaving this in here for context, however.

**Build BOW Representation**

To repeat this for the entire train and test sets, we employ loadData, which uses transfer under the hood. First, we fetch all the absolute paths per the arg passed in. Then, we encode ytrain and ytest using numpy operations. These files are all listed in order, so we can just concatenate arrays of zeroes and ones pertaining to the length of each sub-directory. We then loop through the train and test directories and run the preprocessing/BoW method defined above. I store the resulting representations in a dict, but later convert it to a dataframe so I had visibility of the column names. There was no shuffling employed, so the order of Xtrain is retained in relation to the response variable, and the same holds for the test sets.

```
def loadData(Path, vocab):

    train_pos, train_neg, test_pos, test_neg = fetch_files(
        main_path=Path)  #used fetch_files under the hood

    ytrain = get_y_labels(train_pos, train_neg)  #fetch train labels
    ytest = get_y_labels(test_pos, test_neg)  #fetch test labels

    Xtrain = {}
    for i,j in enumerate(train_pos + train_neg):
        Xtrain[i] = transfer(j, vocabulary=vocab)

    Xtest = {}
    for i,j in enumerate(test_pos + test_neg):
        Xtest[i] = transfer(j, vocabulary=vocab)

    convert_to_df = lambda x: pd.DataFrame.from_dict(x).T
    Xtrain = convert_to_df(Xtrain)
    Xtest = convert_to_df(Xtest)

    Xtrain.columns = vocab
    Xtest.columns = vocab

    return Xtrain, Xtest, ytrain, ytest


```

Listing 3: Loading Data

The resulting four matrices/vectors have the following shapes, where row = document, and column is equal to number of features ($i^{th}$ column represents the $i^{th}$ vocab word, for the X sets. The y sets are just binary flag vectors. Again, these are initially stored as dataframes, and the train process will index on them.:

Xtrain Shape: (1400, 101)
ytrain Shape: (1400,)
Xtest Shape: (600, 101)
ytest Shape: (600,)

**Multinomial Naive Bayes Classifer (MNBC) Training Step**

Onto training a MNBC. This step proved to be much easier than testing. Here we are generating our parameters via MLE. Because this is a binary classification task, we will have two generative models:

$$\Pr(W_1 = n_1, \ldots, W_k = n_k | C = 0) = \Pr(C = 0 | W_1 = n_1, \ldots, W_k = n_k) * \Pr(C = 0)$$

$$\Pr(W_1 = n_1, \ldots, W_k = n_k | C = 1) = \Pr(C = 1 | W_1 = n_1, \ldots, W_k = n_k) * \Pr(C = 1)$$

When we perform MLE on a multinomial distribution, rather than a Bernoulli, we're not only concerned about the presence or lack of presence of a word given a class $c_j$, we're also concerned with the relative frequency of the word appearing in class $c_j$.

$$P\left(w_k | c_j\right) \leftarrow \frac{n_{k,j} + \alpha}{n_j + \alpha |\text{Vocabulary}|}$$

In the above equation, we partition the corpus into positive and negative. For each word in the vocab (14 total here), we count the number of occurrences of that word in each partition and divide it by the total number of words that appear in that partition (non-unique). We repeat for $L$ classes. $\alpha$ acts as our laplacian smoothing so we don't end up multiplying by zeroes when we calculate our posterior.

```
def naiveBayesMulFeature_train(Xtrain, ytrain, alpha = 1):

    pos = Xtrain[:700] #split
    neg = Xtrain[700:] #split
    thetaPos = []
    thetaNeg = []
    for i in pos.columns:
        temp = math.log((pos[i].sum() + alpha) / (sum(pos.sum()) + (alpha * len(pos.columns)))
    ) + np.log(0.5)
        thetaPos.append(temp)
    for i in neg.columns:
        temp = math.log((neg[i].sum() + alpha) / (sum(neg.sum()) + (alpha * len(neg.columns)))
    )+ np.log(0.5)
        thetaNeg.append(temp)

    return thetaPos, thetaNeg

```

Listing 4: Generating Parameters

*Note, this was before dictionary expansion. We can visualize our weights according to each of the generative models learned parameters and see if they match intuition. The first 7 vocab words would generally be characterized as being positive words, while the next five would fall into the negative polarity bucket. For context, if we received an incoming test document only containing the word love, we would classify that as being positive sentiment based on the argmax formulation, where we can transform the product of the evidence into the sum of the log of the evidence because log is monotonically increasing:

$$= \arg\max_{\theta_1 \to \theta_k} \log\left(\prod_{t=l}^{T} \theta_1^{n_{1a}} \theta_2^{n_{di}} \cdot \theta_k^{n_{kd}}\right)$$
$$= \arg\max_{\theta_1^-, \theta_k} \sum_{t-1,j,T} n_{1,d_t} \log(\theta_1) + \sum_{t=1,\ldots T} n_{2,d_t} \log(\theta_2) + \ldots + \sum_{t=1,T} n_{k,d_t} \log(\theta_k)$$

| Word | C = 1 | C = 0 |
|---|---|---|
| love' | (2.754) | (3.264) |
| 'wonderful' | (4.286) | (5.520) |
| 'best' | (2.697) | (3.351) |
| 'great' | (2.777) | (3.660) |
| 'superb' | (5.114) | (6.644) |
| 'still' | (3.074) | (3.410) |
| 'beautiful' | (4.089) | (4.975) |
| 'bad' | (3.482) | (2.667) |
| 'worst' | (5.539) | (4.016) |
| 'stupid' | (5.569) | (4.226) |
| 'waste' | (6.293) | (4.790) |
| 'boring' | (5.402) | (4.068) |
| '?' | (2.059) | (1.879) |
| '!' | (2.967) | (2.584) |

Figure 2: Visualizing Log Proba

After expanding the dictionary to 101 unique tokens, in which most of them are mapped to UNK, hence the larger sum of log probs, we see the top (head and tail) n features for P(C=1). The feature weights aren't as intuitive as in the smaller vocab set.

**Multinomial Naive Bayes Classifer (MNBC) Testing+Evaluate Step**

Onto Inference. We have the ordered data for ytrain and ytest already stored via loadData. We store our learned models (2) in dictionaries for easier lookup. For each of the incoming test instances, we only care about the non-zero instances within each bag of words vector. That list of words is what we'll be using to generate our probability of a class given some distribution. For each of those present words, we extract $\theta$ from the lookup table for each class and add it to temp variables. This is us summing up the log likelihoods for each occurring word. Once all words are looped through, we just need to find the argmax. In the binary case, and to handle ties, I just instituted some basic logic. If the sum of the log likelihoods + log prior for each word in a document is greater for one class than the other, the document is mapped to that class, and vice versa. Ties are handled by mapping to class 1 = Positive.

```
    def naiveBayesMulFeature_test(Xtest, ytest, thetaPos, thetaNeg):
        pos_lu =dict(zip(vocab, thetaPos))
        neg_lu  =dict(zip(vocab, thetaNeg))
        yPredict = []
        for i in range(0, len(Xtest)): #each test instance
            test = dict(Xtest.loc[i])
            words = { k: v for k, v in test.items() if v > 0 }
            pos = 0
            neg = 0
            for i,j in words.items():
                pos += pos_lu[i] * j
                neg += neg_lu[i] * j
            #Thresholds for class mapping
            yPredict.append(np.argmax([neg,pos]))


        Accuracy = np.sum(yPredict == ytest) / len(ytest)
        return yPredict, Accuracy

```

Listing 5: Inference

Figure 3 Shows that this from-scratch model performs on par (even slightly better) than the out of the box sklearn MNBC. 68 percent is not great, but when you think about it, we only used 14 words in our vocab and were able to substantially beat the majority vote baseline. That's why I'm going to go back and use NLTK at the end of this. I imagine using most of the corpus (with some thresholding on token counts) will result in 80-90 percent accuracy on the test set.

```
print('sklearn accuracy:' , naiveBayesMulFeature_sk_MNBC(Xtrain, ytrain, Xtest, ytest) *100, '%')
print('my accuracy:' , naiveBayesMulFeature_test(Xtest, ytest, thetaPos, thetaNeg)[1], '%')
```

executed in 113ms, finished 14:25:29 2019-11-14

```
sklearn accuracy: 68.0 %
my accuracy: 68.33333333333333 %
```

Figure 3: Sklearn vs Scratch

This was then revised to account for the dictionary expansion, resulting in increased test set accuracy (Figure 4). We count the number of occurrences (+ our smoothing) and divide it by the total length of the positive vocab (+smoothing), taking the log, while adding in the log of the prior after. The parameters (thetapos / thetaneg) are identical between both models - See table showing this on page 15. A mistake I was making initially during inference was not including the word counts when summing up the log probabilities. Below accuracies are both 76.5 percent (truncated here).

**Multivariate Bernoulli Naive Bayes Classifer (BNBC)**

There is very little difference in the coding aspect of the MNBC and the BNBC. From the MLE perspective, we're no longer finding the relative frequency of words in a class, however. We no longer care about the multinomial

```
]: #Xtrain, Xtest, ytrain, ytest = loadData(Path = main, vocab = vocab, nltk_pre = False)
   Xtrain, Xtest, ytrain, ytest = loadData(Path= main, vocab=vocab)
   thetaPos, thetaNeg = naiveBayesMulFeature_train(Xtrain, ytrain, alpha =  1)
   thetaPosTrue, thetaNegTrue = naiveBayesBernFeature_train(Xtrain, ytrain)
   print('----------------Multinomial Naive Bayes--------------------------')
   print('my accuracy:' ,np.round(naiveBayesMulFeature_test(Xtest, ytest, thetaPos, thetaNeg)[1],3) *100)
   print('sklearn accuracy:', np.round(naiveBayesMulFeature_sk_MNBC(Xtrain, ytrain, Xtest, ytest),3)*100)
   print('----------------Multivariate Bernoulli Naive Bayes---------------------------')
   print('my accuracy:' , naiveBayesBernFeature_test(Xtest, ytest, thetaPosTrue, thetaNegTrue)[1])
   executed in 27.8s, finished 15:10:05 2019-11-26

----------------Multinomial Naive Bayes--------------------------
my accuracy: 76.5
sklearn accuracy: 76.5
----------------Multivariate Bernoulli Naive Bayes---------------------------
my accuracy: 0.735
```

Figure 4: Sklearn vs Scratch

distribution - We only care if a word is present or not. The inference stage is a bit different, as we just like at the number of documents where a word occurs by the total length of the document set for each class. In the code below, the minor change comes in lines 8 and 12. Rather than count the number of occurrences of a word in all of the documents of a particular class, we only count the number of non-zero occurring documents in a class, for each vocab word. Pandas makes this easy, which is why I went with the dataframe structure here. We still employ smoothing here. Also, instead of dividing by the total sum of occurrences of a word, we only divide by the length of the document set. We increment the numerator by 1 and the denominator by 2 so we don't condition away our probabilities.

$$\hat{p} = \frac{x}{n} \quad \begin{matrix}\text{i.e. Relative} \\ \text{Freq. of} \\ \text{binary event}\end{matrix}$$

$$\mathbb{P}(w_i = true|c) = \frac{\#\text{ files which include } w_i \text{ and are in class } c + 1}{\#\text{ files are in class } c + 2}$$

$$\mathbb{P}(w_i = \text{false}|c) = 1 - \mathbb{P}(w_i = \text{true}|c)$$

```
1  #-------------------------------------------------------------#
2  def naiveBayesBernFeature_train(Xtrain, ytrain):
3      pos = Xtrain[:700] #split
4      neg = Xtrain[700:] #split
5      thetaPosTrue = []
6      thetaNegTrue = []
7
8      for i in pos.columns:
9          temp = ((pos[i].astype(bool).sum(axis=0) + 1) / (len(pos) +
10                                                          2))
11         thetaPosTrue.append(temp)
12     for i in neg.columns:
13         temp = (neg[i].astype(bool).sum(axis=0) + 1) / (len(neg) +
14                                                         2)
15         thetaNegTrue.append(temp)
16
17     return thetaPosTrue, thetaNegTrue
18 #-------------------------------------------------------------#
19 def naiveBayesBernFeature_test(Xtest, ytest, thetaPosTrue, thetaNegTrue):
20
21     yPredict = []
22     for i in range(0, len(Xtest)): #each test instance
23         pos = 0.5
24         neg = 0.5
```

```
        a = np.array(Xtest.loc[i])
        for i in range(0, 101):
            if a[i] > 0:
                pos+= math.log(thetaPosTrue[i])
                neg += math.log(thetaNegTrue[i])
            elif a[i] == 0:
                pos+= math.log(1 - thetaPosTrue[i])
                neg += math.log(1 - thetaNegTrue[i])

        #Thresholds for class mapping
        yPredict.append(np.argmax([neg,pos]))


    Accuracy = np.sum(yPredict == ytest) / len(ytest)
    return yPredict, Accuracy


```

Listing 6: Bernoulli

On the updated vocab of length 101, BNBC runs at a test accuracy of 73.5 percent, so we do lose quite a bit of inference ability when we don't take into account relative frequency. This was validated against Sklearn Multivariate Bernoulli implementation as well. I initially forgot to subtract the log proba for nonoccurring words, but that has been revised here.

————————-Multivariate Bernoulli Naive Bayes————————————
my accuracy: 0.735 sklearn accuracy: 0.735 Below is validation that the model built from scratch is the correct model - The weights/log probas match exactly with those of scikit's implementation.

| Vocab Index | Sklearn MNBC ThetaNeg | Sklearn MNBC ThetaPos | By-Hand MNBC ThetaNeg | By-Hand MNBC ThetaPos | By-Hand BNBC ThetaNeg | By-Hand BNBC ThetaPos |
|---|---|---|---|---|---|---|
| 0 | (12.58) | (14.17) | (12.58) | (14.17) | (1.69) | (2.97) |
| 1 | (15.15) | (13.88) | (15.15) | (13.88) | (3.99) | (2.73) |
| 2 | (11.77) | (11.52) | (11.77) | (11.52) | (1.08) | (0.89) |
| 3 | (14.72) | (14.43) | (14.72) | (14.43) | (3.61) | (3.19) |
| 4 | (13.30) | (15.06) | (13.30) | (15.06) | (2.25) | (3.78) |
| 5 | (13.18) | (13.24) | (13.18) | (13.24) | (2.17) | (2.15) |
| 6 | (14.19) | (13.31) | (14.19) | (13.31) | (3.09) | (2.24) |
| 7 | (13.31) | (12.99) | (13.31) | (12.99) | (2.31) | (1.91) |
| 8 | (13.87) | (13.39) | (13.87) | (13.39) | (2.82) | (2.32) |
| 9 | (13.98) | (13.47) | (13.98) | (13.47) | (3.06) | (2.32) |
| 10 | (13.56) | (13.58) | (13.56) | (13.58) | (2.49) | (2.48) |
| 11 | (13.69) | (13.53) | (13.69) | (13.53) | (2.62) | (2.33) |
| 12 | (13.71) | (13.67) | (13.71) | (13.67) | (2.68) | (2.51) |
| 13 | (13.58) | (13.57) | (13.58) | (13.57) | (2.53) | (2.41) |
| 14 | (13.63) | (13.54) | (13.63) | (13.54) | (2.55) | (2.46) |
| 15 | (13.91) | (13.28) | (13.91) | (13.28) | (2.77) | (2.18) |
| 16 | (13.89) | (13.53) | (13.89) | (13.53) | (2.87) | (2.41) |
| 17 | (13.98) | (13.29) | (13.98) | (13.29) | (3.00) | (2.31) |
| 18 | (13.58) | (13.58) | (13.58) | (13.58) | (2.56) | (2.55) |
| 19 | (14.01) | (13.18) | (14.01) | (13.18) | (2.87) | (2.11) |
| 20 | (13.66) | (13.49) | (13.66) | (13.49) | (2.58) | (2.35) |
| 21 | (13.47) | (13.49) | (13.47) | (13.49) | (2.46) | (2.32) |
| 22 | (13.61) | (13.42) | (13.61) | (13.42) | (2.51) | (2.33) |
| 23 | (14.11) | (13.27) | (14.11) | (13.27) | (2.97) | (2.08) |
| 24 | (13.69) | (13.39) | (13.69) | (13.39) | (2.58) | (2.33) |
| 25 | (13.04) | (13.17) | (13.04) | (13.17) | (2.03) | (2.14) |
| 26 | (12.83) | (13.92) | (12.83) | (13.92) | (1.86) | (2.79) |
| 27 | (13.50) | (12.81) | (13.50) | (12.81) | (2.48) | (1.98) |
| 28 | (13.51) | (13.41) | (13.51) | (13.41) | (2.49) | (2.31) |
| 29 | (14.03) | (13.05) | (14.03) | (13.05) | (2.94) | (2.09) |
| 30 | (13.71) | (13.41) | (13.71) | (13.41) | (2.68) | (2.29) |
| 31 | (13.14) | (13.54) | (13.14) | (13.54) | (2.89) | (2.77) |
| 32 | (13.39) | (13.46) | (13.39) | (13.46) | (2.38) | (2.33) |
| 33 | (13.47) | (13.51) | (13.47) | (13.51) | (2.41) | (2.29) |
| 34 | (13.58) | (13.26) | (13.58) | (13.26) | (2.60) | (2.16) |

Figure 5: model weights

| Vocab Index | Sklearn MNBC ThetaNeg | Sklearn MNBC ThetaPos | By-Hand MNBC ThetaNeg | By-Hand MNBC ThetaPos | By-Hand BNBC ThetaNeg | By-Hand BNBC ThetaPos |
|---|---|---|---|---|---|---|
| 35 | (13.07) | (13.94) | (13.07) | (13.94) | (2.11) | (2.84) |
| 36 | (13.66) | (13.33) | (13.66) | (13.33) | (2.55) | (2.17) |
| 37 | (12.76) | (12.59) | (12.76) | (12.59) | (1.82) | (1.63) |
| 38 | (12.74) | (14.33) | (12.74) | (14.33) | (1.84) | (3.26) |
| 39 | (13.02) | (13.29) | (13.02) | (13.29) | (2.49) | (2.38) |
| 40 | (13.49) | (13.26) | (13.49) | (13.26) | (2.62) | (2.25) |
| 41 | (13.29) | (13.36) | (13.29) | (13.36) | (2.51) | (2.55) |
| 42 | (13.35) | (13.32) | (13.35) | (13.32) | (2.25) | (2.18) |
| 43 | (13.38) | (12.92) | (13.38) | (12.92) | (2.35) | (1.92) |
| 44 | (13.49) | (13.03) | (13.49) | (13.03) | (2.46) | (1.97) |
| 45 | (13.36) | (12.99) | (13.36) | (12.99) | (2.29) | (1.95) |
| 46 | (13.69) | (13.09) | (13.69) | (13.09) | (2.62) | (2.01) |
| 47 | (13.27) | (13.02) | (13.27) | (13.02) | (2.22) | (1.92) |
| 48 | (13.35) | (13.14) | (13.35) | (13.14) | (2.43) | (2.17) |
| 49 | (13.49) | (12.85) | (13.49) | (12.85) | (2.41) | (1.92) |
| 50 | (12.88) | (12.91) | (12.88) | (12.91) | (2.15) | (2.02) |
| 51 | (12.95) | (13.45) | (12.95) | (13.45) | (1.93) | (2.28) |
| 52 | (13.51) | (12.72) | (13.51) | (12.72) | (2.66) | (1.76) |
| 53 | (12.53) | (14.30) | (12.53) | (14.30) | (1.65) | (3.09) |
| 54 | (13.13) | (13.03) | (13.13) | (13.03) | (2.18) | (1.97) |
| 55 | (13.17) | (13.13) | (13.17) | (13.13) | (2.10) | (2.08) |
| 56 | (11.93) | (12.12) | (11.93) | (12.12) | (1.08) | (1.15) |
| 57 | (13.00) | (12.96) | (13.00) | (12.96) | (2.09) | (1.93) |
| 58 | (12.83) | (13.02) | (12.83) | (13.02) | (1.91) | (2.00) |
| 59 | (13.18) | (12.85) | (13.18) | (12.85) | (2.09) | (1.77) |
| 60 | (12.90) | (12.83) | (12.90) | (12.83) | (1.89) | (1.83) |
| 61 | (12.72) | (13.03) | (12.72) | (13.03) | (1.78) | (1.96) |
| 62 | (12.25) | (12.07) | (12.25) | (12.07) | (1.36) | (1.13) |
| 63 | (13.08) | (12.69) | (13.08) | (12.69) | (2.18) | (1.85) |
| 64 | (13.01) | (12.75) | (13.01) | (12.75) | (2.09) | (1.73) |
| 65 | (13.33) | (12.63) | (13.33) | (12.63) | (2.32) | (1.63) |
| 66 | (13.04) | (12.77) | (13.04) | (12.77) | (2.17) | (1.77) |
| 67 | (12.76) | (12.88) | (12.76) | (12.88) | (1.73) | (1.79) |
| 68 | (12.99) | (12.72) | (12.99) | (12.72) | (1.98) | (1.69) |
| 69 | (12.87) | (12.77) | (12.87) | (12.77) | (1.93) | (1.69) |

Figure 6: model weights

| Vocab Index | Sklearn MNBC ThetaNeg | Sklearn MNBC ThetaPos | By-Hand MNBC ThetaNeg | By-Hand MNBC ThetaPos | By-Hand BNBC ThetaNeg | By-Hand BNBC ThetaPos |
|---|---|---|---|---|---|---|
| 70 | (12.68) | (12.71) | (12.68) | (12.71) | (1.69) | (1.69) |
| 71 | (12.60) | (12.61) | (12.60) | (12.61) | (1.65) | (1.54) |
| 72 | (12.99) | (12.67) | (12.99) | (12.67) | (1.96) | (1.65) |
| 73 | (12.64) | (12.67) | (12.64) | (12.67) | (1.76) | (1.59) |
| 74 | (13.06) | (12.51) | (13.06) | (12.51) | (2.09) | (1.54) |
| 75 | (12.17) | (12.20) | (12.17) | (12.20) | (1.27) | (1.17) |
| 76 | (12.48) | (12.74) | (12.48) | (12.74) | (1.58) | (1.70) |
| 77 | (12.26) | (12.46) | (12.26) | (12.46) | (1.39) | (1.44) |
| 78 | (12.66) | (12.58) | (12.66) | (12.58) | (1.85) | (1.72) |
| 79 | (12.81) | (12.67) | (12.81) | (12.67) | (2.01) | (1.89) |
| 80 | (12.84) | (12.27) | (12.84) | (12.27) | (2.08) | (1.56) |
| 81 | (12.49) | (12.46) | (12.49) | (12.46) | (1.80) | (1.58) |
| 82 | (11.81) | (12.25) | (11.81) | (12.25) | (1.01) | (1.19) |
| 83 | (12.04) | (12.58) | (12.04) | (12.58) | (1.13) | (1.52) |
| 84 | (12.04) | (12.20) | (12.04) | (12.20) | (1.37) | (1.42) |
| 85 | (12.28) | (12.41) | (12.28) | (12.41) | (1.59) | (1.53) |
| 86 | (12.18) | (12.25) | (12.18) | (12.25) | (1.34) | (1.32) |
| 87 | (12.41) | (12.09) | (12.41) | (12.09) | (1.58) | (1.27) |
| 88 | (12.08) | (12.12) | (12.08) | (12.12) | (1.24) | (1.19) |
| 89 | (12.06) | (12.05) | (12.06) | (12.05) | (1.24) | (1.18) |
| 90 | (11.81) | (12.11) | (11.81) | (12.11) | (1.14) | (1.21) |
| 91 | (11.86) | (11.46) | (11.86) | (11.46) | (1.05) | (0.71) |
| 92 | (12.17) | (11.54) | (12.17) | (11.54) | (1.32) | (0.88) |
| 93 | (11.17) | (11.26) | (11.17) | (11.26) | (0.61) | (0.58) |
| 94 | (11.94) | (11.53) | (11.94) | (11.53) | (1.15) | (0.78) |
| 95 | (11.78) | (11.59) | (11.78) | (11.59) | (1.06) | (0.88) |
| 96 | (11.46) | (11.53) | (11.46) | (11.53) | (0.78) | (0.79) |
| 97 | (11.18) | (12.25) | (11.18) | (12.25) | (0.72) | (1.34) |
| 98 | (11.36) | (11.30) | (11.36) | (11.30) | (0.70) | (0.61) |
| 99 | (11.03) | (11.12) | (11.03) | (11.12) | (0.55) | (0.57) |
| 100 | (0.00) | (0.00) | (0.00) | (0.00) | (0.00) | (0.00) |

Figure 7: model weights

**Extra Credit**
For extra credit, I ran this through a moderately complex NLTK pipeline involving: punctuation removal, stopword removal, and stemming. I think my implementation is not very scalable, so I had to resort to using sklearn for vectorization and modelling.

```
def nltk_pipe():
    *load data into train and test sets
    Xtrain.columns = ['doc']
    Xtrain['doc'] = Xtrain['doc'].apply(lambda x: ' '.join([word for word in x.split() if word
     not in (stop_words)]))
    Xtrain['doc'] = Xtrain['doc'].apply(preprocess)
    Xtrain['doc'] = Xtrain['doc'].apply(stem)

    *repeat for test set
    vec = CountVectorizer()
    Xtrain = vec.fit_transform(Xtrain['doc'])
    Xtest = vec.transform(Xtest['doc'])
    ytrain = get_y_labels(train_pos, train_neg)  #fetch train labels
    ytest = get_y_labels(test_pos, test_neg)


    return naiveBayesMulFeature_sk_MNBC(Xtrain, ytrain, Xtest, ytest)

```

Listing 7: NLTK Preprocessing Pipeline

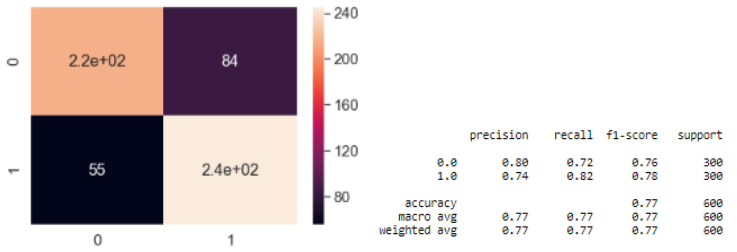
This achieved 82 percent test set accuracy, but was a pretty laborious task as I haven't used NLTK in a while. Rare word removal would have been the only other thing besides lemmatization that I would have liked to employ here.

```
print('NLTK Preprocessing Test Accuracy:', nltk_pipe())
executed in 2.78s, finished 20:16:48 2019-11-14
NLTK Preprocessing Test Accuracy: 0.82
```

Figure 8: Sklearn vs Scratch

Interestingly, if we use n-grams instead of unigrams, we see a steady drop in accuracy:
NLTK Preprocessing Test Accuracy 1-gram: 0.82
NLTK Preprocessing Test Accuracy 2-gram: 0.78
NLTK Preprocessing Test Accuracy 3-gram: 0.67
NLTK Preprocessing Test Accuracy 4-gram: 0.601

**Some additional tweaks**
A look at the confusion matrix of our best model shows that we struggled more with false positives, i.e. predicting positive sentiment when it was actually false. This makes sense, in my opinion, because most of the vocab words had a positive connotation. We show the count distributions (logged count) of the vocab from each of the classes below. There is difficulty in discerning any substantial differences at a glance. UNK heavily skews the resulting log probabilities, however. Last one (Figure 8)- Let's try limiting the number of features on the vectorization (BoW step) before feeding it into the model. We'll use the admittedly better NLTK preprocessing noted above. It is shown that there is a treshold reached where the number of features included in the corpus decays test set accuracy. notably after 4096 unique unigrams, the performance dips and the time to train increases steadily. At 4096, we achieve our best performance in this writeup: 83 percent accuracy on test set.




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0.0 | 0.80 | 0.72 | 0.76 | 300 |
| 1.0 | 0.74 | 0.82 | 0.78 | 300 |
| accuracy | | | 0.77 | 600 |
| macro avg | 0.77 | 0.77 | 0.77 | 600 |
| weighted avg | 0.77 | 0.77 | 0.77 | 600 |

Figure 9: Best Model - Conf Matrix + Class Report

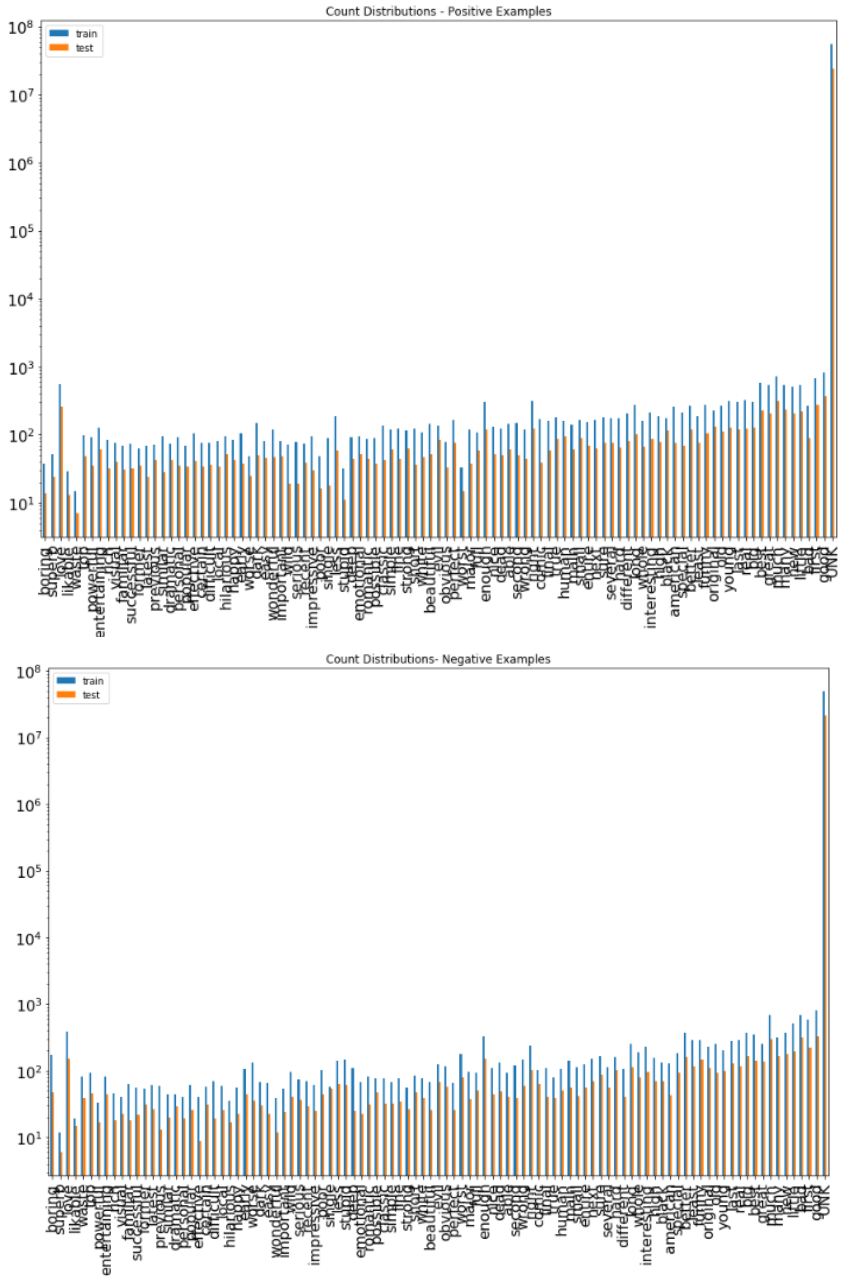



Figure 10: Sklearn vs Scratch + Word Distributions

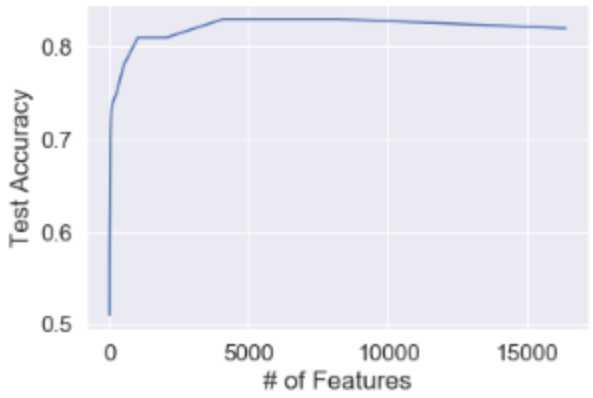



Figure 11: Num of Features Vectorized Against test set accuracy

# 2 Sample QA Questions:

### Question 1. Bayes Classifier

Suppose you are given the following set of data with three Boolean input variables $a, b$, and $c$, and a single Boolean output variable $G$.

| $a$ | $b$ | $c$ | $G$ |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |

For item (a), assume we are using a naive Bayes classifier to predict the value of $G$ from the values of the other variables.

(a) According to the naive Bayes classifier, what is $P(G = 1|a = 1 \wedge b = 1)$?

We want to find the probability of G = 1 conditioned on a = 1 and b= 1. To do so, we will use bayes theorem. The initial equation can be rewritten as:

$$P(G = 1|a = 1, b = 1) = \frac{P(G = 1)P(a = 1|G = 1)P(b = 1|G = 1)}{P(G = 1)P(a = 1|G = 1)P(b = 1|G = 1) + P(G = 0)P(a = 1|G = 0)P(b = 1|G = 0)}$$

P(G = 1) = 1/2
P(G = 0) = 1/2
$P(a = 1|G = 1) = 2/4$
$P(a = 1|G = 0) = 2/4$
$P(b = 1|G = 1) = 2/4$
$P(b = 1|G = 0) = 1/4$
$P(G = 1)P(a = 1|G = 1)P(b = 1|G = 1) = 1/2 * 2/4 * 1/4 = 1/16$

$P(G=0)P(a=1|G=0)P(b=1|G=0) = 1/2 * 2/4 * 2/4 = 1/8$

$$P(G=1|a=1,b=1) = \frac{1/16}{1/16+1/8} = 1/3$$

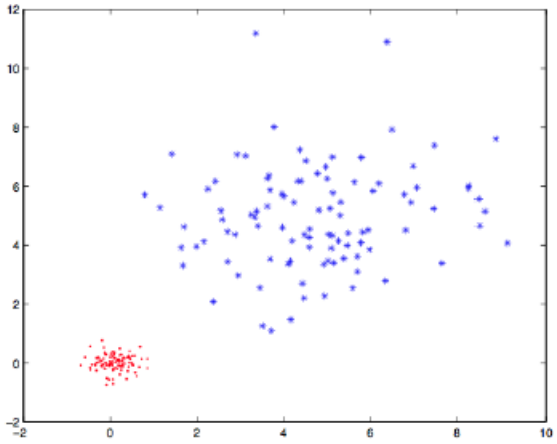
Conversely, the probability of observing G = 0 given a,b = 1 is 1 - the probability computed above:

$$P(G=0|a=1,b=1) = 1 - P(G=1|a=1,b=1) = 2/3$$

Please provide a one-sentence justification for the following TRUE/FALSE questions.
**(b)** (True/False) Naive Bayes Classifier and logistic regression both directly model p(C—X)..
False. Logistic regression, and discriminative classifiers in general, directly model $P(C|X)$ from data. They learn a direct map from input to response. Generative classifiers, like Bayes classifiers, model the joint probability $p(x, y)$ (1, Ng, Jordan).This is also shown in lecture slides 17c.

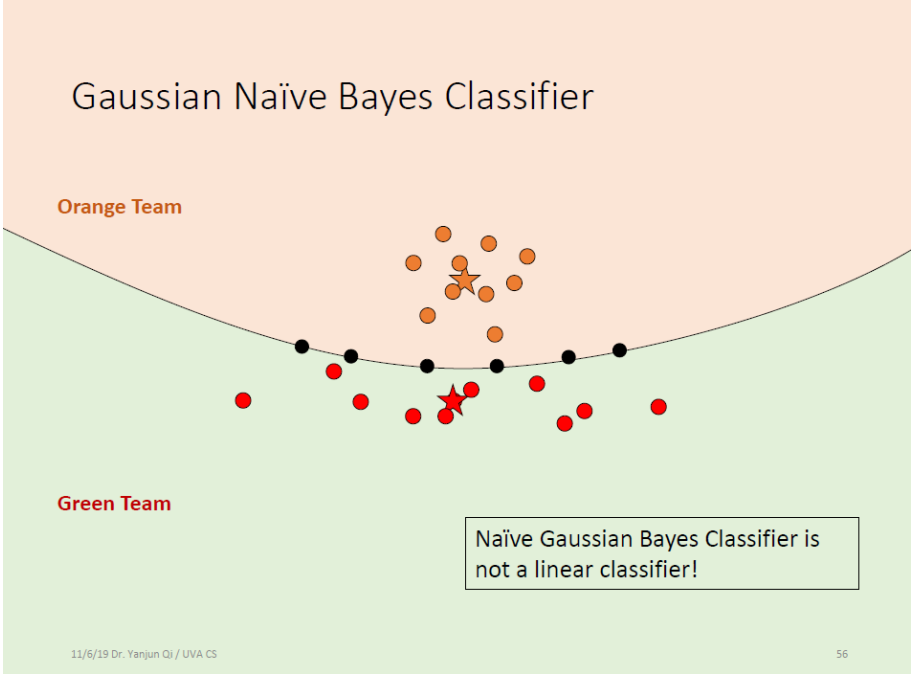
**(c)** (True/False) Gaussian Naive Bayes Classifer and Gaussian Mixture Model are similar since both assume that $p(X|cluster == i)$ follows Gaussian distribution.
More generally, we're looking at different types of learning algorithms. The difference between GMMs and GNBCs is that one is an unsupervised learning method and one is supervised (labels given for GNBCs, not for GMMs). GMMs can be used to probabalistically estimate normally distributed subpopulations within a population [2]. I'm going to say **FALSE** because even though both algorithms have Gaussian assumptions, Naive Bayes classifiers operate with independence assumptions as well, while GMMs, depending on the covariance matrix, can learn different shapes of clusters. A stackoverflow posts references that A GNBC model with diagonal covariance matrices is equivalent to a GMM. [3]. This makes sense, because this GMM could only learn to represent circular clusters where w/ full covariance matrix, which captures linear interactions between features, you can get more of an ellipsis with beliefs.

(d) (**True/False**) when you train Gaussian Naive Bayes Classifier on data samples provided in the following Figure, using separate covariance for each class, $\Sigma_1 \neq \Sigma_2$, the decision boundary will be linear. Please provide a one-sentence justification.




The answer is False, as the decision boundary here will be quadratic if the covariance matrix is not shared between classes. I believe this holds for both the Naive and Non-naive cases (LDA/QDA) Of Gaussian Bayes Classifiers. The below image from the lecture slides can be seen to show this point. See Fig on Page 15 from lecture slides for validation.

**References**
[1] Andrew Y. Ng and Michael I. Jordan. 2001. On discriminative vs. generative classifiers: a comparison of




logistic regression and naive Bayes. In Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic (NIPS'01), T. G. Dietterich, S. Becker, and Z. Ghahramani (Eds.). MIT Press, Cambridge, MA, USA, 841-848.
[2] https://brilliant.org/wiki/gaussian-mixture-model/ [3] https://stats.stackexchange.com/questions/105140/gaussian-naive-bayes-really-equivalent-to-gmm-with-diagonal-covariance-matrices