Raphael Ku
SI 630 HW1

Naïve Bayes Classifier

Part 1

Tokenization Function:

```
def tokenize(string):
    a = string.split()
    return a
```

Training Function:

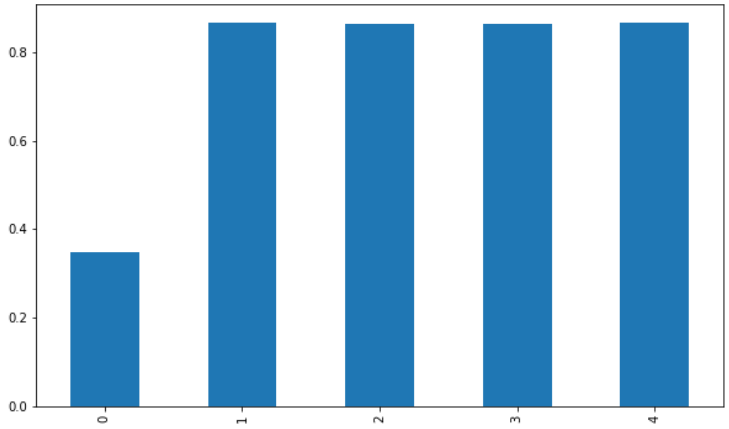
```
def train(tsvfile, smoothing_alpha=0):
    traindata = pd.read_table(tsvfile, sep='\t', quoting=3)
    traindata['tokens'] = pd.Series(map(tokenize, traindata.text))
    voc = defaultdict(lambda: [0,0])
    for x, y in traindata.iterrows():
        for word in y[3]:
            if y[2]==0:
                voc[word][0] += 1
            else:
                voc[word][1] += 1
    vocablength = len(voc)
    neutword = int(sum([voc[word][0] for word in voc]))
    badword = int(sum([voc[word][1] for word in voc]))

    prior = defaultdict(lambda:
[(0+smoothing_alpha)/(neutword+smoothing_alpha*vocablength),(0+smoothing_alpha)/(badword+smoothing_al
pha*vocablength)])
    Py = traindata['class'].value_counts()/traindata['class'].count()

    for word in voc:
        prior[word][0] = (voc[word][0]+smoothing_alpha)/(neutword+smoothing_alpha*vocablength)
        prior[word][1] = (voc[word][1]+smoothing_alpha)/(badword+smoothing_alpha*vocablength)

    return Py, prior
```

Classification Function:

```
def classify(testlst, trained):
    Py, prior = trained
    goodprior = np.exp(sum(map(lambda x: np.log(x), [prior[x][0] for x in testlst])))
    badprior = np.exp(sum(map(lambda x: np.log(x), [prior[x][1] for x in testlst])))
    Px = Py[0]*goodprior+Py[1]*badprior
    x = {0:(Py[0]*goodprior)/Px, 1:(Py[1]*badprior)/Px}
    return max(x, key=x.get)
```

Development data F1: 0.348293

Modification of smoothing_alpha parameter (from 0 as default to 4)



α = 1 was a suitable smoothing alpha, since the graph shows very marginal improvement for increasing the smoothing alpha past one. As I set up a defaultdict object to account for words not in the training data, α = 1 provided the probability mass to these unknown words, leading to significant improvement in the model.

Part 2

Better tokenize function:

```
def better_tokenize(string):
    a = re.sub('(@\w+)|http[s]*[^\s]*', '', string).lower()
    translator = a.maketrans('', '', strg.punctuation)
    a = a.translate(translator)
    return re.findall('[a-zA-z]{3,}', a)
```

Focus of changes for the better tokenization function were the following:

- Remove all mentions (@username) and shortened links (http://t.co/...) as these tend to be 1-ofs in the dataset with minimal predictive value
    - Mentioning a username doesn't indicate anything, shortened links are autogenerated by Twitter, so will almost always be unique

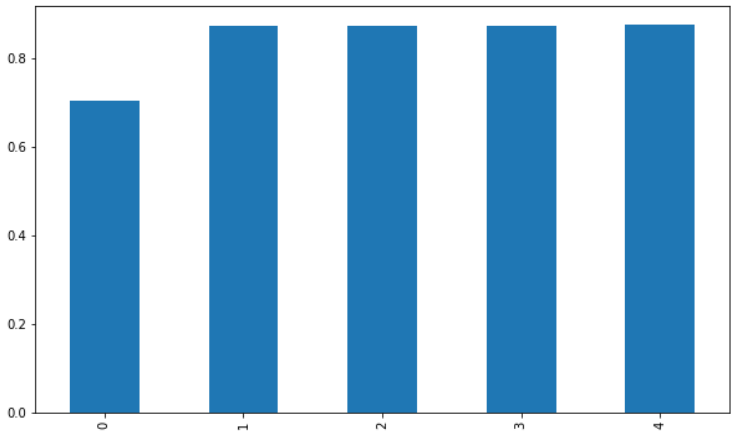
- Punctuation and capitalization issues were resolved (all lower, no symbols)
    - Hashtags are kept as one word, without the hashtag symbol
    - Was made to reduce the number of tokens being processed, as well as making sure formatting was standardized for all words and tokens
    - This also meant removal of emojis. Although they might have some predictive power, I chose to remove them as well due to inconsistencies in encoding (some are Unicode, some aren't) making it difficult to separate them from the words

- Choice to only provide tokens of words with 3 or more characters
    - Meant to filter out words like a, an, and stray letters

Development data F1: 0.702513

Using the better_tokenize function, the Naïve Bayes model showed an improvement of approximately 0.35 versus the whitespace tokenization. This may be due to the reduction in insignificant tokens improving the posterior probability of each individual token still included by reducing the denominator.

Modification of smoothing_alpha parameter (from 0 as default to 4)



Logistic Regression Classifier

sigmoid function:

```
def sigmoid(number):
    return 1/(1+np.exp(-number))
```

log_likelihood function:

```
def log_likelihood(featmatrix, bvector, classvector):
    # Referenced: https://beckernick.github.io/logistic-regression-from-scratch/ >> Full citation in writeup
    score = featmatrix.dot(bvector)
    likelihood = np.sum(classvector*score - np.log(1+np.exp(score)))
    return likelihood
```

compute_gradient function:

```
def compute_gradient(featmatrix, bvector, classvector):
    predictionvector = sigmoid(featmatrix.dot(bvector))
    cost = classvector - predictionvector
    gradient = featmatrix.T.dot(cost).T
    return gradient
```

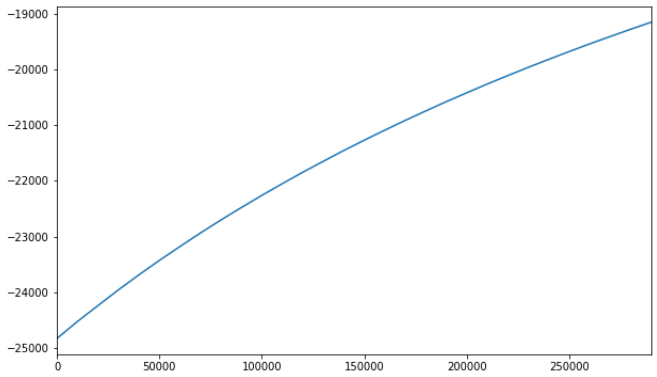
logistic_regression function:

```
def logistic_regression(featmatrix, classvector, learningrate, num_step, intercept=False):
    # Referenced: https://beckernick.github.io/logistic-regression-from-scratch/ >> Full citation in writeup
    weights = np.zeros(featmatrix.shape[1])

    for step in np.arange(num_step):
        if featmatrix.shape[0]-step <= 0:
            i = step % featmatrix.shape[0]
        else:
            i = step
        gradient = compute_gradient(featmatrix[i], weights, classvector[i])
        weights += gradient*learningrate

        if step % 10000 == 0:
            print(log_likelihood(featmatrix, weights, classvector))

    if intercept == False:
        weights[-1] = 0
    return weights
```

predict function:

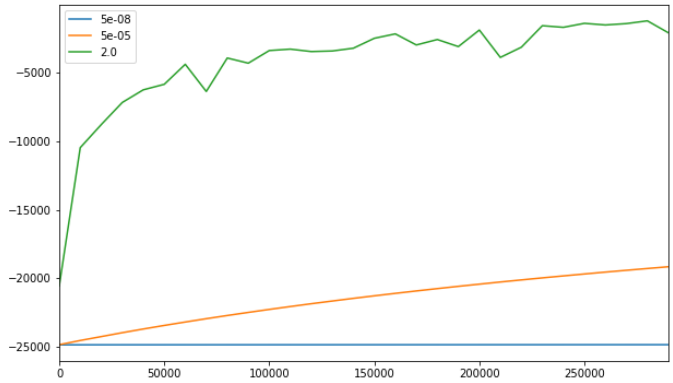
```
def classify(countedtokens, weights):
    if sigmoid(countedtokens.dot(weights)) > 0.5:
        return 1
    else:
        return 0
```

On Training Data and 5e-5 learning rate:



Using 5e-5 as the learning rate, the model failed to converge. What was observed was that log_ likelihood improved by around 300 for each 10,000 steps made. This indicates that the 5e-5 learning rate may be too low given the number of steps provided, given how marginal the improvements were per 10,000 steps, and how my model wasn't converging to a specific value when the program ended.

On Training Data and 3 learning rates
Learning rates used were 5e-8 (smaller), 5e-5 (standard), and 2 (largest)



Based on the plot provided for the 3 learning rates, I can see that the learning rates have a significant effect on the final log likelihood using the trained weights. For the smaller learning rate, due to the low value, the weights barely updated for each iteration, learning to weights that have only minor changes, and provide a similar log likelihood. In comparison, the larger learning rate clearly shows significant improvement in log likelihood, with the biggest change coming from the first 10,000 steps updating the base assumptions for weights. It also seems to be close to stabilizing log likelihood when the 300,000 steps finished. It may indicate that having variable learning rates may be ideal, using large training rates in early iterations to correct the base weights, then switching to a lower learning rate for more stable predictions.

Development data F1: 0.808357

References for code:

Becker, Nick. “Logistic Regression from Scratch in Python.” Nick Becker, 4 Nov. 2016, beckernick.github.io/logistic-regression-from-scratch/.

“Scipy.sparse.csr_matrix.” SciPy v1.0.0 Reference Guide, 25 Oct. 2017, docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html.