Naive Bayes Text Classification

Ana Maria Martinez Sidera
A20410762
Department of Computer Science
Illinois Institute of Technology

February 22, 2018

1. Problem statement

In this assignment you will implement and evaluate the naive Bayes algorithm for text classification. You will train your models on a Imdb reviews dataset of positive and negative movie reviews and report prediction accuracy on a test set.

2. Proposed solution

In this section I will explain the code that I have developed to make this text classification. First, we start from two different databases: one the train (with positive and negative reviews) and the test. First of all we call imdb.py to analyze all the reviews. It returns a csr_matrix where the rows are reviews, the columns are id of each unique words and in each position the number of words we have found in each review.

2.1 Classification and Evaluation

In this section we analyze the data obtained in the train and calculate variables that we will use for the following functions, such as predict label or predict probabilities.

init :

```
self.vocab_len = data.X.shape[1]
self.count_positive = X[np.where(Y == 1)[0], :].sum(axis = 0)
self.count_negative = X[np.where(Y == -1)[0], :].sum(axis = 0)
self.num_positive_reviews = len(Y[np.where(Y == 1)])
self.total_positive_words =  self.count_positive.sum()
self.total_negative_words =  self.count_negative.sum()
self.num_negative_reviews = len(Y[np.where(Y == -1)])
self.P_positive = self.num_positive_reviews / (self.num_positive_reviews
        + self.num_negative_reviews)
self.P_negative = 1 - self.P_positive
self.deno_pos = self.total_positive_words + (self.ALPHA * self.vocab_len)
self.deno_neg = self.total_negative_words + (self.ALPHA * self.vocab_len)
```

Train(self, X, Y) : In this function I calculate the probability that each word is positive or negative. For this I use this function, using a smoothing so that the null values do not affect the final result.

$$P(W|+) = \frac{count\_positive[W] + Alpha}{vocab\_len * Alpha + total\_positive\_word}$$

$$P(W|-) = \frac{count\_negative[W] + Alpha}{vocab\_len * Alpha + total\_negative\_word}$$

With this formula the probabilities are not normalized, so the next step is to normalize them.

PredictLabel(self, X) : In this section we use the probabilities calculated in the previous section and we will give a value of 1 (positive) or -1 (negative) to the test data reviews. To calculate the label we will have to calculate the probability of positive of the review and the probability that it is negative, the highest value will give us the label of the review we are looking for. For this we will use the following formula:

$$P(+) = log(p(+)) + \sum log(p(w_i|+))$$

After performing this operation for all the words that appear in the review if the probability of positive is greater the label of this review is +1. Otherwise, the label will be negative.

```
for i in range(0,X.shape[0]):
    review_words = X.getrow(i).nonzero()[1]
    pos_probs = []
    neg_probs = []
    for index in range(0, review_words.shape[0]):
        current_word_id = review_words[index]
        pos_probs.append(self.prob_positive[current_word_id])
        neg_probs.append(self.prob_negative[current_word_id])
    pos_prob = log(self.P_positive) + np.sum(np.log(np.array(pos_probs)))
    neg_prob = log(self.P_negative) + np.sum(np.log(np.array(neg_probs)))
    if pos_prob > neg_prob:
        pred_labels.append(+1.0)
    else:
        pred_labels.append(-1.0)
```

Eval(self, test) : In the eval function we calculate the accuracy taking into account the cases we have labeled with the prediction and seeing the result we really expected.

```
def Eval(self, test):
    Y_pred = self.PredictLabel(test.X)
    ev = Eval(Y_pred, test.Y)
    return ev.Accuracy()
```

2.2 Probability Prediction

In the equation:

$$p(Y = C|\mathbf{x}) = \frac{p(\mathbf{x}|Y = C)p(Y = C)}{\sum_{k=1}^{|C|} p(\mathbf{x}|Y = C_k)p(Y = C_k)}$$

where we compute the probabilities both the denominator and the numerator can become very small, typically because the p(xi—Ck) can be close to 0 and we multiply many of them with each other. To prevent underflows, one can simply take the log of the numerator, but one needs to use the log-sum-exp trick for the denominator.

LogSum(self, logx, logy) : We compute the next formula:

$$\log \sum_k e^{a_k} = \log \sum_k e^{a_k} e^{A-A} = A + \log \sum_k e^{a_k - A}$$

with:

$$a_k = \log \left(p(\mathbf{x}|Y = C_k) p(Y = C_k) \right)$$

$$A = \max_{k \in \{1,\dots,|C|\}} a_k$$

```
m = max(logx, logy)
return m + log(exp(logx - m) + exp(logy - m))
```

PredictProb(self, test, indexes) : We calculate the probabilities that it is positive and negative taking into account the previous formula used.

$$\log \left(p(Y = C|\mathbf{x}) \right) = \log \left(\frac{p(\mathbf{x}|Y = C) p(Y = C)}{\sum_{k=1}^{|C|} p(\mathbf{x}|Y = C_k) p(Y = C_k)} \right) \tag{1}$$

$$= \log \left(\underbrace{p(\mathbf{x}|Y = C) p(Y = C)}_{\text{numerator}} \right) - \log \left(\underbrace{\sum_{k=1}^{|C|} p(\mathbf{x}|Y = C_k) p(Y = C_k)}_{\text{denominator}} \right) \tag{2}$$

(3)

```
for i in indexes:
    review_words = test.X.getrow(i).nonzero()[1]
    pos_probs = []
    neg_probs = []
    for j in range(0, review_words.shape[0]):
        current_word_id = review_words[j]
        pos_probs.append(self.prob_positive[current_word_id])
        neg_probs.append(self.prob_negative[current_word_id])
    pos_prob = log(self.P_positive) + np.sum(np.log(np.array(pos_probs)))
    neg_prob = log(self.P_negative) + np.sum(np.log(np.array(neg_probs)))
    denominator = self.LogSum(pos_prob, neg_prob)
    predicted_prob_pos.append(exp(pos_prob - denominator))
    predicted_prob_neg.append(exp(neg_prob - denominator))
```

2.3 Precision and Recall

In this section we calculate the accuracy and the recall in data that we have obtained in the test. We have to calculate the confusion matrix of the data obtained in the test. We calculate the probabilities of the data obtained in the test. And we apply a threshold to these results. If the probability that it is positive does not exceed this value, it will be measured as negative. That is, what we modify with the threshold is the position of the vertical bar between predicted 1 and predicted 0. Modifying the values of positives and negatives that we find depending on the threshold that we put between 0 -1. With each

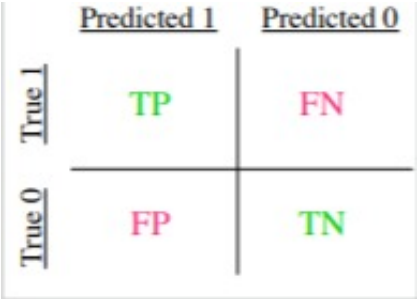

Figure 1: Confusion matrix

confusion matrix we calculate the precision and recall that we will later visualize in the graph.

PredictTreshold(self, test) :

```
predicted_prob = self.PredictProb(test, range(test.X.shape[0]))
    lista = np.arange(0, 1, 0.001)
    precision = [0] * len(lista)
    recall = [0] * len(lista)
    for treshold in lista:
        tresholdlabel = []
        for i in range(len(predicted_prob)):
            if predicted_prob[i] >  treshold:
                tresholdlabel.append(1)
            else:
                tresholdlabel.append(-1)
        conf_arr = self.Curveprecision(test, tresholdlabel)
        precision.append(self.EvalPrecision(conf_arr[0][0],conf_arr[1][0]))
        recall.append(self.EvalRecall(conf_arr[0][0],conf_arr[0][1]))
```

Curveprecision(self, test, tresholdlabel) : We calculate the confusion matrix for all the values.

```
def Curveprecision(self, test, tresholdlabel):
    conf_arr = [[0, 0], [0, 0]]
    for i in range(len(tresholdlabel)):
        if test.Y[i] == 1:
            if tresholdlabel[i] == -1:
                conf_arr[0][1] = conf_arr[0][1] + 1
```

```
            else:
                conf_arr[0][0] = conf_arr[0][0] + 1
        elif test.Y[i] == -1:
            if tresholdlabel[i] == 1:
                conf_arr[1][0] = conf_arr[1][0] +1
            else:
                conf_arr[1][1] = conf_arr[1][1] +1
    return conf_arr
```

EvalPrecision(self, truepositive, falsenegative) : Depending on the values obtained in the matrix confusion, we calculate the precision and matrix values.

$$Precision = \frac{True\_positive}{True\_positive + False\_negative}$$

```
def EvalPrecision(self, truepositive, falsenegative):
    if truepositive > 0:
        return float(truepositive) / (truepositive + falsenegative)
    else:
        return 0
```

EvalRecall(self, truepositive, falsepositive) :

$$Recall = \frac{True\_positive}{True\_positive + False\_positive}$$

```
def EvalRecall(self, truepositive, falsepositive):
    if truepositive > 0:
        return float(truepositive) / (truepositive + falsepositive)
    else:
        return 0
```

2.4 Features

In this function we have to print out the 20 most positive and 20 most negative words in the vocabulary sorted by their weight according to our model. This will require a bit of thought how to do because

- We will need to compute the linear feature weight for each word based on the condition probabilities in our model.
- The words in each document have been converted to IDs and we will need to convert them back.

printtop20(self, X) :

$$a = (log(P(+|w)) - log(P(-|w)))$$
$$positive\_weight = a * count\_positive[word] - count\_negative[word]$$

$$b = (log(P(-|w)) - log(P(+|w)))$$
$$negative\_weight = b * count\_negative[word] - count\_positive[word]$$

```
weights_positive = {}
weights_negative = {}
for i in range(X.GetVocabSize()):
    positive_w = (self.prob_positive[i])/(self.total_positive_words)
    negative_w = (self.prob_negative[i])/(self.total_negative_words)
    weights_positive[i] = (log(positive_w) - log(negative_w))
            *self.count_positive[i]- self.count_negative[i]
    weights_negative[i] = (log(negative_w) - log(positive_w))
            *self.count_negative[i]- self.count_positive[i]
```

For positive words we order the array in a decreasing way to obtain the highest data. We get from each id the word assigned to it and the value.

```
weights_positive = sorted(weights_positive.items(), key=lambda t: t[1])
for wordId, value in weights_positive[::-1][:20]:
    currWord = X.GetWord(wordId)
    currWeight = value
    print (currWord , currWeight)
```

```
weights_negative = sorted(weights_negative.items(), key=lambda t: t[1])
print ('Negative words with weights')
for wordId, value in weights_negative[::-1][:20]:
    currWord = X.GetWord(wordId)
    currWeight = value
    print (currWord , currWeight)
```

2.5 Main

We call all the functions we have used: we do the csr_matrix of the train and test data to be able to use them, we call the NaiveBayes function and we initialize the parameters, we calculate the accuracy, we print the first 10 reviews and we calculate the top 20 positive and negative words.

```
print("Reading Training Data")
traindata = IMDBdata("%s/train" % sys.argv[1])
print("Reading Test Data")
```

```
testdata  = IMDBdata("%s/test" % sys.argv[1], vocab=traindata.vocab)
print("Computing Parameters")
nb = NaiveBayes(traindata, float(sys.argv[2]))
print("Evaluating")
print("Test Accuracy: ", nb.Eval(testdata))
print("Print Precision and Recall curve:")
nb.PredictTreshold(testdata)
(nb.PredictProb(testdata, 10))
nb.printtop20(traindata.vocab)
```

3. Results and discussion

3.1 Classification and Evaluation

The following table lists out the accuracy for various values of the parameter ALPHA.

Table 1: Alpha & Accuracy

Alpha	Accuracy
0.1	0.816
0.5	0.826
1.0	0.830
2.0	0.833
5.0	0.836
10.0	0.837
20.0	0.835

As we can see in the following image, the accuracy increases as we increase the ALPHA parameter until the last value where it begins to decrease. In this model that we have done I see a defect, in the case that the probabilities are equal to doing it with an if and not taking into account that they are equal we will be compensating more one classification than another. We should take this result into account and study it separately in our model. It is not that it is positive or negative, it is that we have the same probability for both. We are making an erroneous assumption.

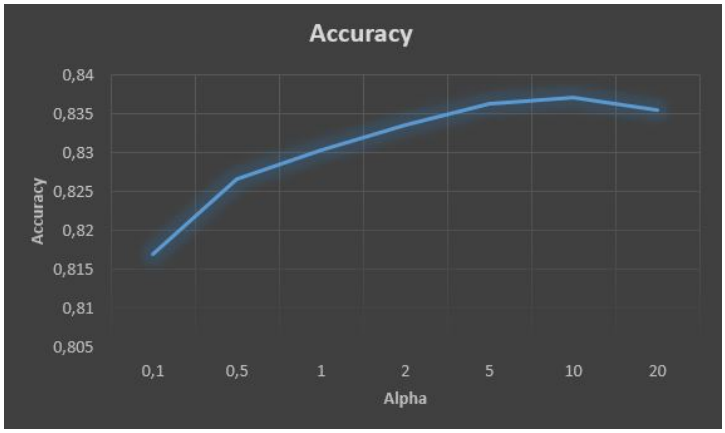

Figure 2: Accuracy graph

3.2 Probability Prediction

In the following table we can see the first 10 reviews. We can see how in all cases we have managed to predict the label of each one. And we see how the probabilities vary in each case, the lowest being the third review. It would make sense to analyze this review in detail to see where we can improve, and then retranslate the model.

Table 2: Probability estimated for the first 10 reviews

Label	Predicted	Probability	Review
-1.0	-1.0	0.9988	I went and saw this movie last night...
-1.0	-1.0	0.9999	Actor turned director Bill Paxton follows up...
-1.0	-1.0	0.7955	As a recreational golfer with some knowledge...
-1.0	-1.0	0.9999	I saw this film in a sneak preview, and it is...
-1.0	-1.0	0.9994	Bill Paxton has taken the true story of the 1913...
-1.0	-1.0	1.0	I saw this film on September 1st, 2005 in Indianapolis...
-1.0	-1.0	1.0	Maybe I'm reading into this too much, but I wonder how...
1.0	1.0	0.9999	I felt this film did have many good qualities...
-1.0	-1.0	0.9999	This movie is amazing because the fact that...
-1.0	-1.0	0.9004	"Quitting" may be as much about exiting a...

3.3 Precision and Recall

Precision-Recall is a useful measure of success of prediction when the classes are very imbalanced. In information retrieval, precision is a measure of result relevancy, while recall is a measure of how many truly relevant results are returned.
The precision-recall curve shows the tradeoff between precision and recall for different threshold. A high area under the curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate. High scores for both show that the classifier is returning accurate results (high precision), as well as returning a majority of all positive results (high recall).

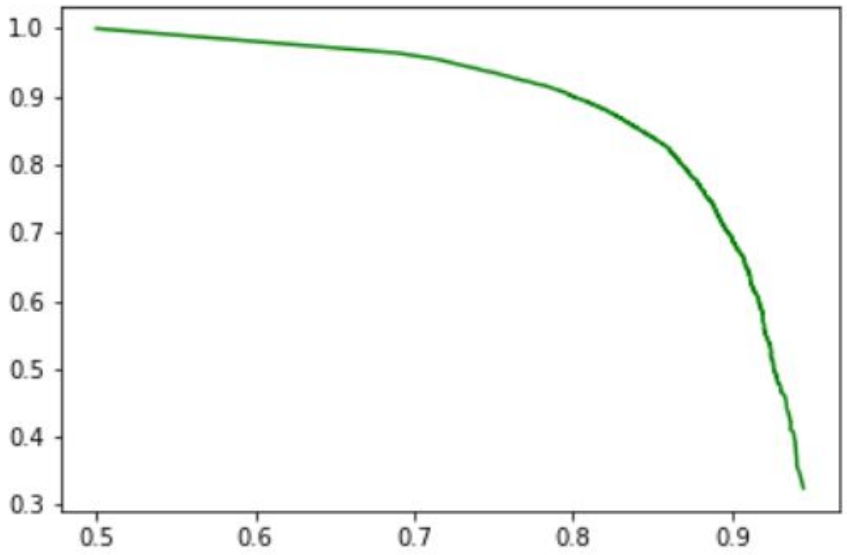

Figure 3: Precision-Recall graph. Alpha = 1.0

3.4 Features

We can observe the words with more weight in the positive documents.

Table 3: The 20 most positive words

Positive word	Weights
great	2546.82
excellent	1467.84
wonderful	1364.30
best	1078.47
loved	836.63
favorite	815.33
!	812.66
amazing	769.22
perfect	686.90
beautiful	675.69
highly	608.86
superb	547.47
fantastic	520.94
beautifully	513.34
wonderfully	505.31
love	478.97
brilliant	453.90
powerful	383.09
enjoyed	372.06
touching	368.86

We have taken into account the number of times that are repeated in positive documents and negative documents. If we had not had the occurrences in each document we would get the most used words: I, a, an the ... In this case, if we remove the most used words in natural language we can get the most used in positive and negative reviews.

Table 4: The 20 most negative words

Negative word	Weights
bad	5609.46
worst	5111.41
waste	3420.30
awful	1479.85
stupid	1418.78
poorly	1331.88
worse	1311.15
bad.	1305.32
nothing	1304.39
supposed	1250.51
poor	1201.33
terrible	1116.17
boring	1105.64
horrible	1027.70
bad,	967.84
minutes	939.63
awful.	903.67
lame	901.19
pointless	782.86
wasted	765.12