

Project 2 Report

Esteban Guillen

Intro

For the second project of the semester we had to classify the famous 20 newsgroup dataset using a Naïve Bayes classifier. This assignment required that we program the algorithms from scratch.

Code Description (see source code for details)

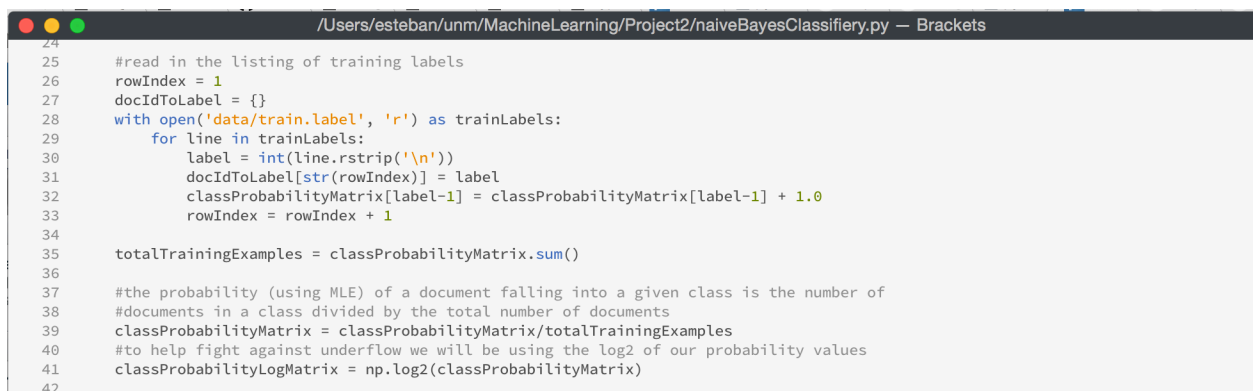
All coding used Python and Numpy was heavily used for linear algebra operations.

Naïve Bayes is very straightforward to implement and our task was made even easier because the data was preprocessed and provided in a convenient format. The code to implement Naïve Bayes had three main parts:

- 1) calculate the class probability $P(Y_k)$ using MLE
- 2) calculate with word probability given the class $P(X_i|Y_k)$ using MAP
- 3) classify new words from the training data into one of the classes

Each part will be briefly described

Calculating the class probability was as simple as counting up the number of documents that were of a given class and dividing by the total number of documents. The log base 2 of the probabilities were used to help with underflow issues.

A screenshot of a code editor window titled "/Users/esteban/unm/MachineLearning/Project2/naiveBayesClassifiery.py — Brackets". The code is in Python and implements the first part of the Naive Bayes classifier: calculating class probabilities. It starts by reading training labels from a file 'data/train.label'. It uses a dictionary 'docIdToLabel' to map document IDs to class labels. A 'classProbabilityMatrix' is updated with counts for each class. The total number of training examples is calculated using 'sum()'. Finally, the class probability matrix is normalized by the total number of examples, and a 'classProbabilityLogMatrix' is created using 'np.log2' to handle underflow issues.

```
24
25 #read in the listing of training labels
26 rowIndex = 1
27 docIdToLabel = {}
28 with open('data/train.label', 'r') as trainLabels:
29     for line in trainLabels:
30         label = int(line.rstrip('\n'))
31         docIdToLabel[str(rowIndex)] = label
32         classProbabilityMatrix[label-1] = classProbabilityMatrix[label-1] + 1.0
33         rowIndex = rowIndex + 1
34
35 totalTrainingExamples = classProbabilityMatrix.sum()
36
37 #the probability (using MLE) of a document falling into a given class is the number of
38 #documents in a class divided by the total number of documents
39 classProbabilityMatrix = classProbabilityMatrix/totalTrainingExamples
40 #to help fight against underflow we will be using the log2 of our probability values
41 classProbabilityLogMatrix = np.log2(classProbabilityMatrix)
42
```

For calculating the word probabilities given the class I first created a matrix to hold the counts of each word by class.

```

/Users/esteban/unm/MachineLearning/Project2/naiveBayesClassifiery.py — Brackets
50 #read through the train data, file is in the format "docId wordId wordCount"
51 with open('data/train.data','r') as trainData:
52     for line in trainData:
53         values = line.rstrip('\n').split(" ")
54         label = docIdToLabel[values[0]]
55         wordId = int(values[1])
56         wordCount = int(values[2])
57         matrixValue = wordCountMatrix[wordId-1,label-1]
58         newValue = matrixValue + wordCount
59         #build the word count matrix (being careful with the indexes)
60         #the wordCountMatrix is needed to calculate the wordProbabilityMatrix below
61         wordCountMatrix[wordId-1,label-1] = newValue
62         wordToDocumentCount[wordId-1] = 1

```

Then I created the word probabilities given the class using the following equation:

$$P(X_i|Y_k) = \frac{(\text{count of } X_i \text{ in } Y_k) + \beta}{(\text{total words in } Y_k) + \beta * (\text{length of vocab list})}$$

```

/Users/esteban/unm/MachineLearning/Project2/naiveBayesClassifiery.py — Brackets
65 #create the wordProbabilityMatrix (probability of a word given the class)
66 #the wordProbabilityMatrix will have shape [numberOfWordsInVocabulary X numberOfClasses]
67 vocabSize = numberOfWordsInVocabulary
68 for v in range(0,numberOfClasses):
69     totalWordsInClassV = wordCountMatrix[:,v].sum()
70
71     for w in range(0,numberOfWordsInVocabulary):
72         wordId = int(w)
73         label = int(v)
74         countOfwInClassV = wordCountMatrix[w,v]
75         #Using MAP estimates to calculate probabilities
76         probabilityOfwGivenV = ((countOfwInClassV) + beta) / (totalWordsInClassV + beta*vocabSize)
77         wordProbabilityMatrix[wordId,label] = probabilityOfwGivenV
78
79 #again we will use the log2 of the probabilities to help fight against underflow
80 wordProbabilityLogMatrix = np.log2(wordProbabilityMatrix)

```

For the classification step I first created a matrix to hold all the testing samples and their word counts. Next I read the test labels ('real' categories) into a dictionary. Then I read the testing samples into the matrix.

```

/Users/esteban/unm/MachineLearning/Project2/naiveBayesClassifiery.py — Brackets
130
131 #matrix will hold the test data and will be used in the classification operation
132 testMatrix = np.zeros((numberOfTestingExamples,numberOfWordsInVocabulary))
133
134 #read through the test labels and save to a dictionary, values will be used to calculate accuracy
135 rowIndex = 1
136 docIdToTestLabel = {}
137 with open('data/test.label', 'r') as testLabels:
138     for line in testLabels:
139         docIdToTestLabel[str(rowIndex)] = int(line.rstrip('\n'))
140         rowIndex = rowIndex + 1
141
142 #read through the test data and populate the testMatrix with word counts
143 with open('data/test.data','r') as testData:
144     for line in testData:
145         values = line.rstrip('\n').split(" ")
146         label = docIdToTestLabel[values[0]]
147         docId = int(values[0])
148         wordId = int(values[1])
149         wordCount = int(values[2])
150         testMatrix[docId-1,wordId-1] = wordCount

```

After the "testMatrix" is created I took the dot product of it with the word probability matrix. Then I added in the class probability matrix. The code was motivated by the following equation:

$$Y^{new} = \operatorname{argmax}[\log_2(P(Y_k)) + \sum_i (\# \text{ of } X_i^{new} * \log_2 P(X_i|Y_k))]$$

```

152
153 #part of classify calculation
154 # Sum of Xi * log2(P(Xi|Yk)) -
155 classifySumMatrix = np.dot(testMatrix, wordProbabilityLogMatrix)
156
157 #finishing out the classify calculation, adding in the Class probability P(Yk)
158 classifyProbabilityMatrix = classifySumMatrix + classProbabilityLogMatrix
159

```

The classify matrix now contains the probability of each test document belonging to each class. The class with the highest probability value is the prediction. The real label value was looked up and compared to the prediction to calculate accuracy. A confusion matrix was also built using predicted and real label values for each testing sample.

```

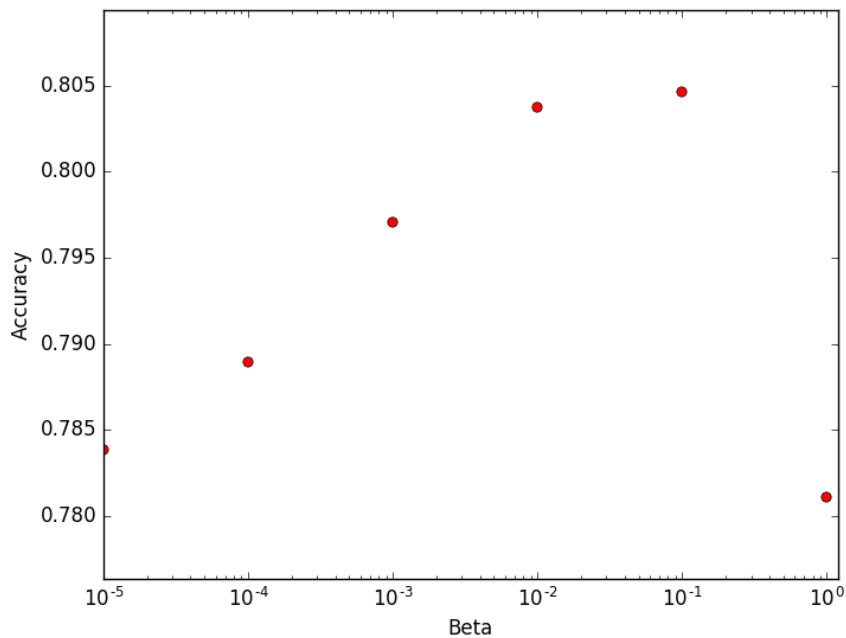
160
161 #confusionMatrix to show a nice visualization of our classification accuracy
162 confusionMatrix = np.zeros((numberOfClasses,numberOfClasses))
163 errorCount = 0
164 correctCount = 0
165 for e in range(0,numberOfTestingExamples):
166     #the prediction for each word is the index + 1 (to account of zero indexing) of the max value of the row
167     prediction = np.argmax(classifyProbabilityMatrix[e,:]) + 1
168     realLabel = docIdToTestLabel[str(e + 1)]
169     confusionMatrix[realLabel-1,prediction-1] = confusionMatrix[realLabel-1,prediction-1] + 1
170     if prediction != realLabel:
171         errorCount = errorCount + 1
172     else:
173         correctCount = correctCount + 1
174

```

Accuracy vs Beta

We were asked to vary beta between .00001 and 1.0 then report the results. The table and graph below summaries the results:

Beta	Accuracy
0.00001	78.4 %
0.0001	78.9 %
0.001	79.7 %
0.01	80.4 %
0.1	80.5 %
1.0	78.1 %



An explanation for the results is given in the answer to Question 4 below.

Questions

- 1) If we can't assume conditional independence there would simply be way too many probabilities to calculate. You would never be able to classify 1000 documents with a vocabulary of 50,000 words.
- 2) The confusion matrix is shown below. Accuracy was 78.5%

```

Project2 — bash — 132x25
Estebans-MacBook-Pro:Project2 esteban$
Estebans-MacBook-Pro:Project2 esteban$ python naiveBayesClassifiery.py

Beta: 1.6343073805321303e-05
Accuracy: 0.7852098600932712
[[ 249.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.  2.  0.  3.  3. 24.  2.  3.  4. 26.]
 [ 0. 286. 13. 14. 9. 22. 4. 1. 1. 0. 1. 11. 8. 6. 10. 1. 2. 0. 0.]
 [ 1. 33. 204. 57. 19. 21. 4. 2. 3. 0. 0. 12. 5. 10. 8. 3. 1. 0. 5. 3.]
 [ 0. 11. 30. 277. 20. 1. 10. 2. 1. 0. 1. 4. 32. 1. 2. 0. 0. 0. 0.]
 [ 0. 17. 13. 30. 269. 0. 12. 2. 2. 0. 0. 3. 21. 8. 4. 0. 1. 0. 1.]
 [ 0. 54. 16. 6. 3. 285. 1. 1. 3. 0. 0. 5. 3. 6. 4. 0. 1. 1. 1.]
 [ 0. 7. 5. 32. 16. 1. 270. 17. 8. 1. 2. 0. 7. 4. 6. 0. 2. 1. 2.]
 [ 0. 3. 1. 2. 0. 0. 14. 331. 17. 0. 0. 1. 13. 0. 4. 2. 0. 0. 1.]
 [ 0. 1. 0. 1. 0. 0. 2. 27. 360. 0. 0. 0. 3. 1. 0. 0. 1. 1. 0.]
 [ 0. 0. 0. 1. 1. 0. 2. 1. 2. 352. 17. 0. 1. 3. 3. 5. 2. 1. 5. 1.]
 [ 2. 0. 1. 0. 0. 0. 2. 1. 2. 4. 383. 0. 0. 0. 0. 1. 2. 0. 1.]
 [ 0. 3. 0. 3. 4. 1. 0. 0. 0. 1. 1. 362. 2. 2. 2. 0. 9. 0. 5.]
 [ 3. 20. 4. 25. 7. 4. 8. 11. 6. 0. 0. 21. 264. 9. 7. 1. 3. 0. 0.]
 [ 5. 7. 0. 3. 0. 0. 3. 5. 4. 1. 0. 1. 8. 320. 8. 7. 6. 5. 8.]
 [ 0. 8. 0. 1. 0. 3. 1. 0. 1. 0. 1. 4. 6. 5. 343. 3. 2. 1. 12.]
 [ 11. 2. 0. 0. 0. 2. 1. 0. 0. 0. 0. 0. 2. 0. 362. 0. 1. 2. 15.]
 [ 1. 1. 0. 0. 0. 1. 1. 2. 1. 1. 0. 4. 0. 5. 2. 1. 303. 5. 23.]
 [ 12. 1. 0. 1. 0. 0. 1. 2. 0. 2. 0. 2. 1. 0. 0. 6. 3. 326. 18.]
 [ 6. 1. 0. 0. 1. 1. 1. 0. 0. 0. 0. 5. 0. 10. 6. 2. 63. 6. 196.]
 [ 39. 3. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 1. 0. 2. 6. 27. 10. 3. 7. 151.]]

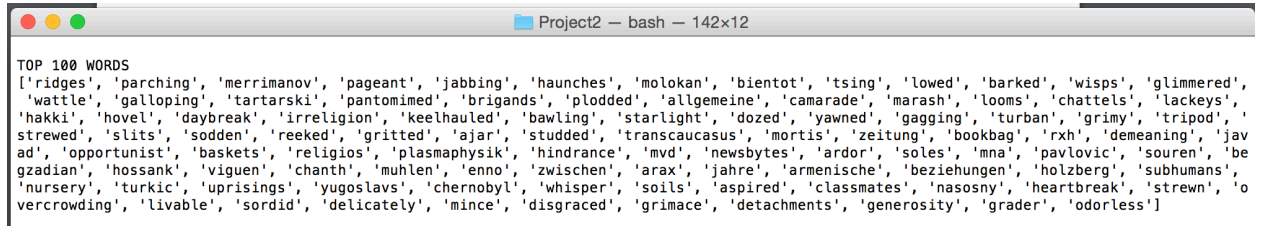
```

- 3) The classifier struggled with the following categories: comp.windows.x, sci.electronics.talk.politics.misc, and talk.religion.misc. I think the problems is that these categories had other similar categories (sci.electronics and comp.sys.mac.hardware for example) so the classifier would have trouble picking between the multiple choices since the topics are similar (they would have similar text across training and testing samples).
- 4) The results show that accuracy is not as good for really small and large beta values. I think this is expected. We add in beta values to avoid zero word probabilities. Some of the words that have zero counts in the training data might show up a number of times in the testing data. So really small beta values would help avoid zero probabilities but would give those words effectively zero influence on classification. As beta grows larger these would start to have an impact on classification and we see accuracy increase. On the other hand, if beta gets too big you are over smoothing and giving some words that shouldn't have a significant impact on the classification a larger contribution. As a result, accuracy will start to go down.
- 5) To identify the words that my classifier relied on the most I used information gain. The following equation (source: <http://www.time.mk/trajkovski/thesis/text-class.pdf>) was implemented (lines 83-128 in my code):

$$\begin{aligned}
 I(w) = & - \sum_{i=1}^k P_i * \log(P_i) \\
 & + F(w) \sum_{i=1}^k p_i(w) * \log(p_i(w)) \\
 & + (1 - F(w)) \sum_{i=1}^k (1 - p_i(w)) * \log(1 - p_i(w))
 \end{aligned}$$

Where P_i is the class probability, $F(w)$ is the probability that a word is found in a document, and $p_i(w)$ is the probability of a word given the class.

- 6) My top 100 words are printed below and most of them are words that I would expect.

A terminal window titled "Project2 — bash — 142x12" displays a list of 100 words. The text is as follows:

```
TOP 100 WORDS
['ridges', 'parching', 'merrimanov', 'pageant', 'jabbing', 'haunches', 'molokan', 'bientot', 'tsing', 'lowed', 'barked', 'wisps', 'glimmered',
'wattle', 'galloping', 'tartarski', 'pantomimed', 'brigands', 'plodded', 'allgemeine', 'camarade', 'marash', 'looms', 'chattels', 'lackeys',
'hakki', 'hovel', 'daybreak', 'irreligion', 'keelhauded', 'bawling', 'starlight', 'dozed', 'yawned', 'gagging', 'turban', 'grimy', 'tripod',
strewed', 'slits', 'sodden', 'reeked', 'gritted', 'ajar', 'studded', 'transcaucasus', 'mortis', 'zeitung', 'bookbag', 'rxh', 'demeaning', 'jav
ad', 'opportunist', 'baskets', 'religios', 'plasmaphysik', 'hindrance', 'mvd', 'newsbytes', 'ardor', 'soles', 'mna', 'pavlovic', 'souren', 'be
gzadian', 'hossank', 'viguen', 'chanth', 'muhlen', 'enno', 'zwischen', 'arax', 'jahre', 'armenische', 'beziehungen', 'holzberg', 'subhumans',
'nursery', 'turkic', 'uprisings', 'yugoslavs', 'chernobyl', 'whisper', 'soils', 'aspired', 'classmates', 'nasosny', 'heartbreak', 'strewn', 'o
vercrowding', 'livable', 'sordid', 'delicately', 'mince', 'disgraced', 'grimace', 'detachments', 'generosity', 'grader', 'odorless']
```

- 7) The words in my top 100 are representative of the language you would see in an online newsgroup but they might not be representative of the type of language you would find in a college text book or the text of a conversation. So trying to classify text outside of the online newsgroup domain would probably see poor accuracy. For text classification to work well in the general case you would need sources of training data from as many different types of distributions (books, newspapers, magazines, etc.) as you can find.