John Alexander

jta00@umail.ucsb.edu

8327934

Summary: My implementation is very similar to a mathematically standard naïve Bayes classifier using plus-one smoothing and tokens consisting of one, two, and three-word sequences from the current document. It achieves an accuracy of 82.6% using the provided training and testing data sets, though it performs somewhat better (in the 85% range) when the training and testing data sets are shuffled together and the training and testing sets are selected randomly but in the same proportions as the original sets.

Architecture: My code consists of two classes: one for running the program and tokenizing the input and one for building and applying the naïve Bayes model. They are called, respectively, "DecisionTreeClassifier" and "BayesModel." The DecisionTreeClassifier class reads input from the specified files, tokenizes the inputs, creates an instance of BayesModel, inserts the tokenized training reviews into the BayesModel, and then loops over the training and testing sets, querying the constructed model and checking the correctness of its output. Besides its default constructor, the BayesModel class has two public methods: one for inserting an ArrayList of strings representing a single review into the model and one for classifying a review (also represented as an ArrayList) as positive or negative. Internally, the model consists of two Hashtable objects with string keys and integer values, as well as two integers representing the number of positive and negative reviews that have been entered into the model. Each Hashtable records the number of times that a particular key has occurred in either a positive or negative document (one Hashtable for each case).

Preprocessing: Before processing, both input strings are scrubbed for HTML tags since some of the input review contain "
" tags. Each review is then split using the regex "[^\w\-']|(\-\-)" which matches all characters that are not letters or numbers or (-) or (') in addition to (--). The tokenizer then removes all non-alphanumeric characters from the start and end of the word. At this point, if the string is empty, it is ignored. Finally, the tokenizer ignores the word if it is part of a short pre-defined list of stop words, which is defined as follows:

```
static String[] sw =
    {"I", "a", "about", "an", "are", "as", "at", "be", "by",
        "for", "from", "how", "in", "is", "it", "of",
        "on", "or", "that", "the", "this", "to", "was",
        "what", "when", "where", "who", "will", "with", "the"};
```

If the words survives all of these tests, it is returned by the tokenizer and added to an ArrayList of words in the current review. This ArrayList represents the features of the document. The full

set of features is all one, two, and three-word runs in the document, which can easily be extracted from the ArrayList.

Model Building: A call to BayesModel object's "addToModel" method contains an ArrayList representing a tokenized article and a boolean indicating whether the corresponding review is positive. As the method loops over the ArrayList, it increments the occurrence count for the current word and its space-separated concatenation with the previous one and two words (if they exist) in the positive or negative Hashtable, depending on the boolean method parameter. At the end of the method, the count of either positive or negative documents is incremented. When an article is being classified, the probabilities of its tokens are multiplied together in the standard way for a naïve Bayes learner, except that keys that have occurred in fewer than 3 documents are excluded, which helps to prevent rare and effectively random shared substrings from corrupting the data.

Results: The algorithm produces 82.6% accuracy on the provided testing data set after training on the training set, as well as 96.65% accuracy on the testing dataset. If these sets are shuffled together and new sets with the same proportions are selected, the algorithm performs somewhat better (85% range), indicating that it is does not overfit the provided data and has good generality. With regard to time, building the model takes slightly over a second, and classifying all of the testing data (500 reviews) takes a little over one fifth of a second. The most significant words for each class in terms of the ratio of one class vs. the other are given below, along with the corresponding ratio.

Positive:

1. star wars 38.0
2. recommended 27.0
3. brosnan 26.0
4. kinnear 24.0
5. pierce 23.0
6. spirits 23.0
7. chess 22.0
8. show people 21.0
9. stardust 19.0
10. davies 19.0

Negative

1. laughable 30.0
2. prom 29.0
3. prom night 27.0
4. sandler 25.0
5. sucks 23.0

6. tedious 19.0
7. pile 19.0
8. scarecrow 17.0
9. garbage 16.5
10. worst 16.0

Challenges: I initially encountered difficulty in properly tokenizing the input strings, particularly with regard to removing the appropriate punctuation. I would have preferred to keep somewhat more punctuation (for example, possessive plurals) as this could communicated important information, but this was not possible in the given time for various reasons. For example, the previous example requires fairly advanced grammatical analysis. I eventually resorted to removing all punctuation before and after a word and keeping only internal punctuation. I also had to decide whether to exclude some of the tokens from consideration based on infrequent occurrence or a very close occurrence value to the other sentiment. To establish the optimal values, I created two loops that varied these two parameters and found the most accurate values. The results were inconsistent, but they ultimately indicated that no results should be excluded based on similarity in occurrence to the other sentiment and that some words that occurred infrequently should be excluded. However, the minimum number of occurrences to exclude varied between 3 and 20.

Weaknesses: I would have liked to experiment with more advanced preprocessing systems, including stemming, to establish whether they would improve accuracy or not. Even the elimination of “ing” and “es” from the ends of words might improve overall performance. Furthermore, I did not fully test the effectiveness of different values in additive smoothing and used add-one smoothing, the simplest form. Further testing in this area could be useful. Finally, when evaluating which of these options to take, a larger training set would have been necessary. I felt limited by the size of a testing set since missing only a few additional tests could cause 2+% change in accuracy, which is significant when optimizing parameters.