CSE 498 Text Mining

Fall 2016

Project 1

Sachin Joshi

Task 1

The extracted reviews of many products in the category of musical instruments from Amazon are stored in the text file named **review.txt** and the rating of each respective review is stored in the text file **rating.txt**. This information is extracted from the dataset **Musical_Instruments_5.json** with the help of the function that has been already provided. The complete reviews along with their corresponding ratings are stored in the file named **reviews.csv**.

I have used R programming language to partition the data into two subsets namely training and test datasets and to discretize the ratings into class 1 and class 0 based on the ratings of the particular review. The R file named **hw.R** contains the code for the above mentioned tasks. We need to install the R programming environment so as to execute this file. However the file can be opened in any of the text editors such as Notepad++ in order to review the code. After running the program, the desired results have been categorized into 4 different files:

1. **NegativeReviews.csv**: This file contains those reviews that have a negative rating (1-3) and belong to class 0.
2. **PositiveReviews.csv**: This file contains those reviews that have a positive rating (4-5) and belong to class 1.
3. **TrainingDataset.csv**: This file contains randomly partitioned data that contribute to approximately 80% of the entire dataset and belongs to the training dataset.
4. **TestingDataset.csv**: This file contains randomly partitioned data that contribute to approximately 20% of the entire dataset and belongs to the testing dataset.

The Naive Bayes categorization system predicts whether the reviews of the products in the category of musical instruments from Amazon are positive or negative. It classifies strings of words to categories using Bayes rule. I have used Java programming language to classify the reviews into positive and negative categories. The Java file to depict this functionality is **NaiveBayesClassifier.java**. Other supporting Java files are stored in the package **saj415**.

Summary

My implementation is very similar to a mathematical standard Naive Bayes classifier using 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.

Working

Baye's rule states that P(x|y), the probability of x occuring given that y has occured, is equal to P(y|x)P(x)/P(y).

Let **s** be the string which we would like to classify. Let **w1,w2,...,wn** be the words in s. Let c be an arbitrary category.

P(c|s) = P(s|c) * P(c) / P(s) = [P(w1|c)P(w2|c)...P(wn|c)] * P(c) / [P(w1)P(w2)...P(wn)]

Our aim is to calculate **P(c|s)** for all categories **c**, and classify **s**, in whichever **c** maximizes **P(c|s)**.

Accuracy and Results

After randomly selecting 80% of the data (training data) to train with, the performance is tested on the remaining 20% of the data (test data). The accuracy is found out to be approximately 83%.

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 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 takes a little over four fifth of a second.

	Testing Data (20%)	**External Testing Data**	**Training Data Set (80%)**
Accuracy (%)	**83**	**97**	**99**

- **External testing data:** Accuracy is approximately 97%
- **Testing on the training set:** Accuracy > 99%

Task 2

To accomplish this task I have used Java programming language. Initially the reviews are tagged using the Stanford POS Tagger. However, I have used only the first 100 reviews to show the functionality of this task. The first 100 reviews are tagged using the Stanford POS Tagger and stored in the file **Tagged_100_Reviews.txt**. However, I have tagged all the reviews using the Stanford POS Tagger and stored in the file **Tagged_Reviews.txt**.

The Java file **AttributesExtract.java** contains the code for extracting the attributes from the text file **Tagged_100_Reviews.txt**. This Java program reads the text file **Tagged_100_Reviews.txt**, extracts the attributes, prints the extracted attribute and stores the extracted attributes in a new text file named **Attributes_Extract.txt**. We must create a new text file and rename it as **Attributes_Extract.txt** before running our program to store the extracted attributes.

Note: All the above mentioned files, i.e., **AttributesExtract.java (Java Program), Tagged_100_Reviews.txt (Input Text File)** and **Attributes_Extract.txt (Output Text File)** must be stored in the same folder.

To compile the above Java file, we use the following syntax in the command prompt:

javac AttributesExtract.java

Once the file is successfully compiled, a .class file named **AttributesExtract.class** is created and we can run the program using the following syntax in the command prompt:

java AttributesExtract

The next step is to count the frequency of each extracted attribute. To accomplish this task, I have designed a simple Java program to count the frequency of each extracted attribute in an input file. The associated Java file is **CountAttributesFrequency.java** and the input file is **Attributes_Extract.txt**.

In this program, there are two classes-**Counter** and **CountAttributesFrequency** class. In the Counter class, two data structures are implemented- **LinkedList** and **TreeMap**. The Counter class has four methods. The first method, **readWords()** is called to read the contents from the input file and split the content into words. Before adding these words to the LinkedList, the regular express is used to filter words by removing all symbol characters from the words. The **Pattern** class is used to define the pattern to be matched. The string pattern "\\W+" matches any symbol except underscore in a word. The **Matcher** class is able to remove the symbols from the words that match the string pattern. The second method is called **countWords()**. This method uses two loops to process all words and count the word frequency. The **TreeMap** is used to store the unique words and their frequencies. The words are stored automatically in **TreeMap**. The **addToMap** method is called by the **countWords()** method to add words and frequencies to the TreeMap. The **showResult** method is invoked after the words and frequencies are added to the TreeMap to show the table of the words, frequencies, and the percentages and further to write the results to two different text files namely **Attributes.txt** and **Frequency.txt**. **Attributes.txt** file contains the unique attributes without any repetition and the **Frequency.txt** file contains the frequency of the corresponding attribute.

The final method, **processCounting** combines the methods above in a single code block. This method will be invoked from the CounterAttributesFreqency class to start analyzing the content

of the input file and displaying the word frequency table. Finally the attributes and the frequency of each attribute is stored in separate text files namely **Attributes.txt** and **Frequency.txt respectively**.

The program can be compiled using the following syntax in the command prompt:

javac CountAttributesFrequency.java

Once the above Java file is compiled, we can run the program using the following syntax in the command prompt. We need to provide the input text file **Attributes_Extract.txt** as the command line argument while running the program as described below:

java CountAttributesFrequency Attributes_Extract.txt

Finally, the desired results of this task, i.e., the top 50 attributes with highest frequency are stored in the file named **Poject_1_Task_2.csv**.

Task 3

The Java file named **AttributesOpinionsExtract.java** is used for extracting the attributes-opinion pair from the text file **Tagged_100_Reviews.txt**.

The program reads the **Tagged_100_Reviews.txt** text file, extracts the attributes-opinion pairs, prints the extracted attributes and opinions and stores them in a new text file named **Attributes_Opinions_Extract.txt**.

Note: We must create a new text file and rename it as **Attributes_Opinions_Extract.txt** before running our program to store the extracted attributes and opinions for each review.

All the above mentioned files, i.e., **AttributesOpinionsExtract.java (Java Program), Tagged_100_Reviews.txt (Input Text File)** and **Attributes_Opinions_Extract.txt (Output Text File)** must be stored in the same folder.

To compile the above Java file, use the following syntax in the command prompt:

javac AttributesOpinionsExtract.java

Once the file is successfully compiled, a .class file named **AttributesOpinionsExtract.class** is created and we can run the program using the following syntax in the command prompt:

java AttributesOpinionsExtract

The final results of this task containing the attributes-opinions pairs having one of the top 50 attributes are stored in the file named **Project_1_Task_3.csv**.