Introduction to Data Mining

Assignment 1

The project was broken into many parts, and this report gives a short description on the approach and the workings of those parts.

1. **Fetch documents from website and store data in a convenient data structure**

   Http GET requests are sent to retrieve each .sgm file. The html received as part of the response is parsed and the topics, places, id and body of each article is stored in a list of objects where each object corresponds to an article. Each artcle has the class labels topics and places. We also added an id for the article which we extracted from an attribute of the REUTERS tag called oldid as we found that many articles did not possess topics or places or had duplicate topics and places.

2. **Parsing the body of each article**

   The body of each article was tokenized to access individual words in the article. RegexTokenizer was used for this purpose. A data structure was created to keep track of the words in an article. Python dictionary data structure was used. The key would be the words, which make the articles. The value was the count of that word in that article.

   We also used a global dictionary to store every word in every article and the global word count. We utilized an ordered dictionary (a high performance collection in python) for this as we would need to sort this list later based on frequency.

   Each token was examined to see if it was already present in previous article or in the same article. If it was already present, the count of the word was increased both on the local dictionary for that article and also the global dictionary for the corpus. If the word is new, then the word would be added to both the local dictionary and global dictionary. During this process, the commonly occurring stop words like ‘the’, ‘there’ etc was removed. This leads to a massive reduction in processing.

   The words were stemmed before adding it to the dictionary. There are many methods available to stem a given word. We experimented with few of them. Lancasterstemmer is currently being used, but we are open to experiment with other stemmers in the future.

3. **Filtering**

   As mentioned earlier, an ordered dictionary is used to store the global dictionary of words and their associated counts. The words are sorted on the basis of the global word count, and the first 95.5% of the words are removed. It was found that there were too many words that has very small global frequencies, and in retaining the top 5% we made sure that we used only the 1053 important words as our attributes for our feature vector. We also eliminated the most common word (which happened to be 'said').

   We didn't use any intelligent filtering mechanism, we just relied on global frequencies. We didn't group together synonyms of words as a single word. We did stem however.

4. **Feature vector**

   Our feature vector used the filtered words as attributes. There are 1053 words and hence 1053 dimensions to our feature vector. Each dimension of the feature vector was the number of times that attribute or word appeared in the corresponding article. Our vector was fairly sparse. We wrote the feature vector to a file for easy viewing. Topics and places were used as class labels. We also displayed the id of each article. We displayed every word chosen as an attribute on standard output along with the global frequency of that word.

Conclusion

We believe that using tf-idf scores on the filtered words to populate the feature vector's dimensions might be a better metric of how important that word is to that document. We can also investigate whether using bigrams or trigrams has any impact on processing speed or the relevance of an attribute to a particular article. Using term frequency is a simple metric that can be used to determine the feature vector of a document. Extending our program to use tf-idf scores is straightforward.