SD-TSIA214 Machine Learning For Text Mining - Lab 2

Sentiment Analysis

PART ONE - Classifier Implementation

The goal of this TP is to implement a Naive Bayes classifier to perform sentiment analysis on movies, trying to predict if they will have a positive or negative review. Our dataset is composed by 2000 review documents, each one labelled either as positive or negative.

1. Complete the count_words function that will count the number of occurrences of each distinct word in a list of string and return vocabulary (the python dictionary. The dictionary keys are the different words and the values are their number of occurences).

In the *count_words* function implementation, there are fundamentally two choices: get immediately all distinct words present in the collection of texts and then update their count, or start with an empty vocabulary and update it each time a new word is encountered. The second approach was followed. The function, in fact, starts with an empty vocabulary and starts counting words: if the current word is already known, it updates the count of this word in the current text; if the word is not already known, it appends it at the end of current's text count array and adds a new rule to the vocabulary, corresponding to the same index. In the end, zero padding has to be applied to arrays to return a NxN matrix. With respect to the returned **vocabulary** dict, I found a little contrast between this question's specifications on the subject and the comments at the beginning of the method provided. The specifications at the beginning of the method were followed, so vocabulary is a dictionary containing known words as key and as value the index that corresponds to the position of word count in the counts array.

2. Explain how positive and negative classes have been assigned to movie reviews (see poldata.README.2.0 file)

According to what is written in the file, reviews have a positive label if they have a rating higher or equal to 3.5 stars out of 5 or 4 out 5, and a grade not lower than B. Reviews have a negative review if their rating is lower than 2/5 stars, or 1.5/4 stars, or a grade equal or lower than C.

3. Complete the NB class to implement the Naive Bayes classifier

Complete the NB class means to carry out the implementation of *fit* and *predict* methods.

The *fit* method computes the probabilities for each class (which is the **prior** probability) simply as Nc = number of texts having class c / N = all texts and the probabilities for each word w conditioned to class c (which the **conditional** probability), representing the probability of having word w in a text knowing that its class is c.

The *predict* method carries out the predicting task by computing a score for each class c: the class with the higher score will then be choosen as the prediction. The score of each class is initialized to the prior probability of that class, and then incremented by the logarithm of the conditional probability for that class of each word present in the text to predict.

4. Evaluate the performance of your classifier in cross-validation 5-folds.

```
With stop words...
Cross Validation 5-fold score: 0.8215 mean 0.015049916943292404 std
```

5. Change the count_words function to ignore the “stop words” in the file data/english.stop. Are the performances improved ?

```
Without stop words...
Cross Validation 5-fold score: 0.8275 mean 0.008366600265340763 std
```

It seems that not using stop words helps - even if by little - in increasing accuracy. In both cases, an accuracy higher than 0.8 is a very satisfying result.

PART TWO - Scikit-Learn Use

1. Compare your implementation with scikitlearn.

```
--- Simple Count Vectorizer (counting words) ---
Cross Validation 5-fold score: 0.812 mean 0.012884098726725092 std

--- Count Vectorizer with char analyzer ---
Cross Validation 5-fold score: 0.6094999999999999 mean 0.018261982367749664 std

--- Simple Count Vectorizer with ngrams ---
Cross Validation 5-fold score: 0.6115 mean 0.014456832294800962 std
```

Experimenting with CountVectorizer options seems not to help accuracy results, which are at their highest when the Count Vectorizer is used with default settings. This happens probably due to the fact that text data is not always well defined, as seen while testing: lots of sentences contain formatting / web characters (i.e. '\n'), which may lead to unconsistent representations with characters and ngrams.

What is impressive is that our implementation of the Naive Bayes classifier was able to obtain better results (0.82 vs 0.81) than the scikit-learn implementation!

Obviously, in terms of time (and, consequently, of efficiency) the scikit-learn implementation vastly outperforms ours.

2. Test another classification method scikitlearn (ex : LinearSVC, LogisticRegression).

The chosen classifier is a LinearSVC,

```
Cross Validation 5-fold score: 0.833 mean 0.01623268308074792 std
```

Among all the classifier used/implemented in this lab, the LinearSVC is the one with the best accuracy, which is better than "my" Naive Bayes classifier's one and also than the scikit MultinomialNB's.

3. Use NLTK library in order to process a stemming. You will used the class SnowballStemmer.

```
--- Stemming - My Naive Bayes Classifier ---
Cross Validation 5-fold score: 0.8220000000000001 mean 0.013546217184144048 std

--- Stemming - LinearSVC ---
Cross Validation 5-fold score: 0.8164999999999999 mean 0.00902773504263389 std
```

4. Filter words by grammatical category (POS : Part Of Speech) and keep only nouns, verbs, adverbs and adjectives for classification.

```
--- Stemming and POS Filtering - My Naive Bayes Classifier ---
Cross Validation 5-fold score: 0.8210000000000001 mean 0.009027735042633867 std

--- Stemming and POS Filtering - LinearSVC ---
Cross Validation 5-fold score: 0.812 mean 0.016462077633154326 std
```

Unfortunately, using stemming seems to not improve our classifier performances. Also filtering words by their POS doesn't increase accuracy.

About the implementation, it has to be said that it is higly inefficient, as confirmed by extremely long run times: the *stemText* method stems and filters words one by one and, in the overall process, we check two times for stop words (both in *stemText* and *count_words* methods).