University of Bergen

INFO284

Group Assignment 1: Naive Bayes Classifier for Sentiment Analysis

Candidates: 110 and 21

Spring 2018

Contents

1 Detailed description of our classifier

1.1 Extracting the reviews

The program in this assignment has been written from scratch. Initially we implemented our own ways of extracting the data from the files, but as we became more familiar with Python we found many useful packages that already perform many of these tasks. In order to be able to classify anything, we must first extract the relevant data from the dataset. In this case, movie reviews in plaintext. We need to extract every single word from these reviews. The movie reviews are user-submitted reviews from the website IMDB.com, which contain some HTML.

To find all the txt files we ask on startup for the path to the directory containing the test and train folders; this is saved as `path.config` for future use. To find all the reviews we made the function `get_filelist` which takes the path to the directory as an argument and scans the directory for files that contains `.txt`, the path to all those files are then appended to a list for further use.

To avoid extracting HTML such as `</br>` which is scattered around many of the reviews, we must carefully remove characters that we do not want to affect our classifier while making sure we do not alter the actual review itself. To achieve this, we made the function `remove_characters` which takes a path (to the TXT-file), an empty list and alternatively a string of text, as arguments. It will then open the text file and convert all characters to lowercase and store it as a local variable. We do this in order to avoid multiple versions of the same word. We want to know the total number of times *"good"* appears, not *"Good"* or potentially *"GOod"*. Furthermore this function removes various characters that could affect how the different words are weighted, it also skips words that are just 1 character, as they can't have any effect on whether or not a review is positive or negative.

If a hypothetical review is *"This movie was great!"*, the word *"great"* would be weighted differently depending on if the exclamation mark was attached or not attached. The following characters are removed from the text: \'()/<>.":,!?. Initially we just removed the characters and handled HTML markup after, but this would cause issues where new words were created. Some reviews in such large datasets are guaranteed to not be written properly, and these cases, although probably not that many should be handled accordingly. With our inital implementation a review like

i liked the movie.it was very good!anyone else wish it lasted longer?I sure did,but it's fine.

would end up like this:

['liked', 'the', 'movieit', 'was', 'very', 'goodanyone', 'else', 'wish', 'it', 'lasted', 'longerI', 'sure', 'didbut', 'it's', 'fine']

Instead of just removing the character we add a space:

['liked', 'the', 'movie', 'it', 'was', 'very', 'good', 'anyone', 'else', 'wish', 'it', 'lasted', 'longer', 'sure', 'did', 'but', "it's", 'fine']

After removing unwanted characters from the review, we split up all the words. Each word is checked if it is *br* (HTML linebreaks), if it isn't it will be appended to the final list of words that was given as argument. We now have a *bag of words.*

1.2 Preparing the classifier

Once all the reviews has been extracted from the training data, we need to assign some score to the words that indicates what weight they should have. This is done with the function `count_text` which takes a list of words as argument and returns a dictionary with the words as keys and their frequency as the value. When preparing our classifier we give the list of all the words from the positive reviews as argument, followed by the negative reviews.

Next step is to find the prior probability. This is simply done by counting the number of positive and negative reviews. We can then add these together and divide the total number of reviews by the number of positive and negative reviews. In both the training data and the test data, both the number of positive and negative is equal, so the prior probability is 0.5 - there is an equal chance of a review being positive or negative.

To avoid having to do this for every classification we save this as a file `classifier.trained`.

1.3 Preparing the test data

In order to test our classifier we need to locate and extract the test data. Similarly to how we extract the training reviews we go through the `pos` and `neg` folders (in the training folder) and retrieve the path to all the `.txt` files. Followed by going through each review, removing all unwanted characters and adding the words to a list. Each list is a review. Once the data is processed it will be saved as a file `test.dataset` to improve performance later. Once the test data is prepared we must count the number of times the different words occur in the review, this will make it simpler to calculate the score of the reviews.

1.4 The classification

Once the words have been counted, each review will be classified. To classify the review we built the function `classify` which takes the review (the counted words, as a dictionary) as an argument, as well as the optional argument `use_stop_words` which by default is `False`. `classify` will then make two calls to another function, `make_class_prediction` which is where the "magic" happens, once for the negative value and once for the positive value. This function takes the list of words (the review), the positive or negative word-frequency dictionary, the prior probability (0.5), the number of reviews and the optional boolean value `use_stop_words` which is `False` by default.

The initial prediction is declared as `1`. We summarize all the values from the word frequency list and add the number of reviews – this will be used later. Then we will go through each word in the review, look up in the dictionary of counted words (the review) how many times the word appears in the review, multiply this with how many times it appears in the positive or negative reviews, add 1 and divide by the summarized values we mentioned earlier. We then take the `log` of this value and add it to prediction. When looking at other implementations it appears that multiplication is used instead of just adding the value from each word. This caused underflowing issues for us, so we opted to go with the addition instead.

Once we have gone through every word in the review, the prediction will be multiplied by the prior probability.

We make the final decision when both the positive and negative value have been calculated. If the negative value is greater than the positive value, `-1` is returned. If the positive value is greater, `1` is returned. So `-1` means it is a negative review, `1` means it is a positive review.

2 Performance

2.1 Efficiency

Parsing through 50,000 different text files is a time-consuming process. In order to speed this up, we save the training and test data as files, `classifier.trained` and `test.data`. This makes it possible to run new tests, for example one with stop words, or do other work without the classifier having to go through all the text files again. This is a massive performance increase, greatly reducing the time needed to run classifications. After running it once, the next times it will only take about 0.04 seconds to load the classifier and 0.50 seconds to load the test dataset. Instead of the 71 seconds and 69 seconds it took to process the training data and testing data on the first run.

2.2 Accuracy

Our classifier has an accuracy of 82.29%, meaning we have an error rate of 17.71%. To increase accuracy, we used the Python package `stop-words` (`https://pypi.org/project/stop-words/`). This package gives us the opportunity to filter out common English words that contain no sentiment value. Using stop words means we remove a great deal of "empty" words, which slightly increases the precision of the model when running a prediction. Running the test with stop words enabled, gives us an accuracy of

about	being	doing	herself	only
above	below	didn't	himself	other
after	between	doesn't	how	over
and	both	during	here	own

Examples of stop words

83.92%, comparing it to the earlier 82.29%, we can see that this is a slight improvement. To get an understanding of why this works we dug in to the training data to see if there is any clear signs of differences in the positive and negative data that could cause this. One interesting thing we discovered is that in the training data there is a significant difference in the total length of the lists, containing all the words from the positive and negative reviews. The positive reviews' length is `2806716`, indicating that there are in total 2.8 million words that we have extracted. Looking at the negative reviews we can see that this number is quite a bit lower, `2728082` is the length of the list containing all the data from the negative reviews. This shows that there are `78634` less words in the negative reviews. This could be a sign that people are likely to spend more time on a review that is positive, and therefore write a longer review – maybe summarizing the movie?

There were about 2.47% more words in the positive reviews, and the accuracy of the classifier was about 2% better when using the stop words. So our hypothesis about positive reviews being longer – and thus, probably have more of these common words that do not indicate any sentiment, is backed up by our accuracy results when using stop words.

Out of curiousity, we made it possible to let the classifier be trained by the testing data as well, and classify either the training data or the testing data, for example letting the classifier predict the training data, having trained it with the test data. It is also possible to train and predict with the same set. The accuracy predicting the training data set using the test set for training is at 83.52% and 84.66%, without and with stop words, respectively. We

noticed that running the classifier with stop words enabled *always* reduced the error rate, which would imply that the stop words are just noise which negatively affect the classifier.

2.3 Improvement

The positive reviews contain a higher amount of word than the negative reviews on average. Adding word count of a review as a feature into the equation could then possibly improve accuracy, at the cost of efficiency. Tinkering with the default values for when words are not found in the positive or negative vocabulary can impact the results. Using 1 instead of 0 caused the accuracy of some tests to increase by 0.1%. In other tests it was reduced by up to 2%. We ended up leaving the default value as 0, 1 is always added before dividing so we will never divide by 0.

Implementing threading where it makes sense would improve the efficiency of the model. Currently, we do not use threads and Python only consumes about 15% of the available CPU power(1 of the 8 logical cores on our test machine). This could also ensure that the model would perform better if run at a much bigger scale.

Using numpy arrays instead of simple dictionaries and lists could possible improve performance as well.

2.3.1 Natural Language Problems

Natural language contains so-called *n-grams*, where one word in a sequence of words could change the meaning of some of the other words. For example, the sequence *"not bad"* has a neutral or possibly a positive meaning. However, the model separates these two words into single words, and we are left with *"not"*

```
not  was found  14173  times in the positive reviews

not  was found  16219  times in the negative reviews
```

and *"bad"*.

```
bad  was found  1860  times in the positive reviews

bad  was found  7233  times in the negative reviews
```

The original sequence of words no longer has the same positive sentiment. An improvement towards a similar problem would be to use lemmatization, which is to group together all words that essentially have the same meaning, into one single word. For example, consolidating *"going, to go, have gone"* into *"go"*. This would also reduce the total amount of words which in turn would also increase both accuracy and efficiency of the model.

3 Results

Why do we get the results we have? One example is the count of the word *"movie"*. This word is counted 24,115 times in the negative reviews, but only 18,436 times in the positive reviews. The word count of the positive reviews is considerably higher than the negative word count, with 77,800 more words in the positive reviews. Which could indicate that users that enjoyed a movie would be more inclined to write a more comprehensive review than someone who did not enjoy the film. Or it is also possible that the vocabulary is simply smaller amongst the negative reviewers.

The word *'good'* is an example of an ambiguous word that appears almost the same amount in positive as in negative.

```
Type in the word: good

good  was found  7454  times in the positive reviews

good  was found  7210  times in the negative reviews
```

One could expect that these kind of words would appear more often in either of the review sets. Ironically, *'positive'* appears with the highest frequency in the negative files.

```
Type in the word: positive

positive  was found  184  times in the positive reviews

positive  was found  326  times in the negative reviews
```

4 Model generalization

Our model is not tailored specifically to text based on movie reviews; it does not take use of any domain specific words, or filter out features based on certain words.

Apart from the fact that the program parses the files based on the specific folder structure of the data set, the classifier would most likely work on other sentiment analysis data sets that require a negative or positive classification (although this is yet to be tested). The `stop-words` package would also work identically with other text files.

5 Our environment

Our program was tested to run on Windows 10, using Anaconda. But should work on any OS, with the standard Python 3.6 release. We use one package that is not part of Anaconda which is `stop-words` and can easily be installed to your environment:

```
pip install stop-words
```

When it comes to performance, our measurements come from running the tests on a PC with the following specifications:

CPU:	Intel i7 4790k @4.20GHz
Storage:	120gb SSD 500MB/s R/W
RAM:	16GB DDR3 @1333MHz

We recommend using an SSD to increase performance.