Data Warehousing and Data Mining
COMP9318: Project Report

Due on Sunday, May 27, 2018

Joshua Long & Christopher Pollock

Introduction

This report details the process of forming a text modification algorithm designed to fool a target classifier into misclassifying a set of test examples. The provided data for this task contained short paragraphs of text that fall into two classes, class 0 and class 1. The two training files ”class-0.txt” and ”class-1.txt” provided examples from each of these two classes while the test data file ”test_data.txt” contained class 1 examples only. The purpose of our text modification algorithm was to modify this class 1 test data in order to fool the target classifier into predicting the data as class 0. The exact selection of parameters and other implementation details for the target classifier was not known. This meant there was an element experimentation as well as trial and error in determining the optimal text modification algorithm.

Understanding the Data

The first task in forming our text modification algorithm was to understand the data and work out what the two classes represent. After visual inspection of the two training files ”class-0.txt” and ”class-1.txt” the data appeared to be indistinguishable to human eyes.

In a further attempt to characterise the classes, we ran some tests to find the relative frequency of words in each class. The results did not provide any meaningful information to assist us, at this point we speculated that each class may in fact be slightly different distributions of words that do not fall into any natural categories.

Training a Classifier

After we had investigated the data, our next goal was to train an accurate local classifier to be used for information extraction and local testing. After learning how to implement a SVM classifier using sklearn, we next started running tests on the data with a linear kernel on its default settings. This approach when using all provided training examples proved to not give accuracy any better than chance when tested on the test_data.txt file.

It soon became clear to us that in order to evaluate the accuracy of our classifier, we would need test examples for both class 0 and class 1. Since we were only given a test file for class 1 and we had twice as many training examples for class 0, we used half the examples in ”class-0.txt” for training, and the other half for testing. This gave us 180 training examples for each class, 180 test examples for class 0, and 200 test examples for class 1.

After further research, we learned about text frequency-inverse document frequency (tf-idf), which improved the classifier. This seemed intuitive at this point as applying tf-idf meant we were assigning much less value to words such as ”the” which appear in every training example. By using this term-weighting scheme we were able to achieve accuracy percentages in the high sixties. We now at least knew that there was a detectable statistical difference between the classes, although we still could not determine what they represented.

From this point we did some experimentation to improve our classifier accuracy with different kernels and performed a grid search to find optimal parameters. However, this did not yield much improvement. With a linear kernel, we were unable to improve on the default parameters. Our best polynomial kernel was only slightly better.

Fooling the Target Classifier

After training our local classifier, the next task was to find a way to extract useful information from it and utilise this information in our text modification algorithm. The most intuitive approach was to extract feature weights from a linear SVM and use these weights to form a priority structure. We recognised that these weights correspond to the importance the SVM assigns to each feature (word) when predicting the class. A large positive weight means that the word strongly suggests class 1, and a large negative weight strongly suggests class 0. A weight near 0 means that the word has little effect on the classification.

After extracting these feature weights we formed a priority structure that would dictate which modifications our algorithm would choose to make. It made sense to place the features in decreasing order of their absolute weight value. Our most successful implementation did this by dividing the features into two groups, one with negative weights associated with class 0 and the other with positive weights associated with class 1. Highest priority for modifications was given to the features at the beginning of each list which contained the most negative or the most positive weights respectively. Our submissions experimented with slight variations on this concept and are detailed below.

Submission 1

Our first idea to fool the target classifier was simply to substitute the 10 words in each example that most strongly suggest class 1 with the 10 words which most strongly suggest class 0. Our algorithm worked like this:

- Train a classifier on the examples in class-0.txt and class-1.txt.
- Extract the weights that the classifier assigned to each word.
- Create a list of all the class 0 words, sorted in decreasing order of the absolute value of their weight. i.e. the words that most strongly suggest class 0 are at the start of the list.
- Create a dictionary mapping each class 1 word to an integer rank. A low rank means high importance, so that when we sort by rank, the most important words come first.
- For each example in test_data.txt, find the 10 most important class 1 words and substitute them with the 10 most important class 0 words that are not already present. If the example does not contain 10 class 1 words, add extra words to total 20 changes.

This worked well when tested against our own classifier. We realised that the target classifier would be trained against more examples, and would differ from ours in unknown ways, and would therefore assign different importances to each word. However, we hoped that it would be similar enough for our algorithm to be fairly effective.

We did our first submission using this algorithm, and achieved 48% accuracy.

Submission 2

The next attempt varied from the first by sorting the words into one priority list and selecting the top 20 addition or removal modifications based on the absolute feature weight value. This approach stemmed from the observation that many of the words we were removing in the first submission did not have a high feature weight according to our local classifier. The idea was that it would be more effective to add more class 0 words with high feature weights than remove the weaker class 1 words with low feature weights.

The algorithm for our second submission performed as follows:

- Train a classifier on the examples in class-0.txt and class-1.txt.
- Extract the weights that the classifier assigned to each word.
- Create a combined list of all the class 0 and class 1 words, sorted in decreasing order of the absolute value of their weight.
- For each example mark whether a word is present or not. Start at the beginning of the sorted combined list created in the previous step and use the following conditions to decide on a modification.
    - If the feature is weighted towards class 1 and is present in the example then remove it.
    - If the feature is weighted towards class 0 and is not present in the example then add it.
    - Ignore all other cases and stop once 20 modifications have been made.

The algorithm favoured adding class 0 words since there was typically less than 5 words present in each example with high class 1 feature weights, at least according to our own classifier. This approach looked promising, but when we submitted we only achieved 18% accuracy.

Submission 3

Our third attempt returned to the word swapping algorithm of our first attempt, but this time we used a TfidfVectorizer with binary=True rather than tf-idf. The resulting accuracy was 33.5%, worse than our first attempt, so we decided that this was a dead end.

Submission 4

We made a number of changes for our fourth attempt. We noticed that the CountVectorizer removed punctuation, e.g. (mr. changed to mr). Since the word ”mr” had a high weight in favour of class 1, we thought it might be better to leave punctuation in place. We read the documentation on CountVectorizer, and learned that you can use a custom tokenizer to change its default behaviour. We modified our code to use a custom tokenizer which only splits on spaces, and does not remove punctuation. Interestingly, this showed that parentheses were one of the most important features for distinguishing the classes. We verified that parentheses are about twice as frequent in class 0 as class 1. This change also improved the accuracy of our own classifier to around 70%.

We also revisited the observation that adding more words than removing had produced poor results, and decided to try only removing words. In some cases our code might not find 20 class 1 words to remove, in which case it would add some class 0 words. When we submitted, our accuracy improved to 92%.

Submission 5

We next decided to verify whether or not tokens such as ”(”, ”)”, and ”,” really are useful for classification. We modified our algorithm slightly so that it would not consider these tokens for removal. This caused our success rate to fall to 78.5%, showing that punctuation seems to be one of the important differences between the classes.

Submission 6

Our best results so far had come from only removing words, but we wanted to see if we could find an optimal ratio of removals to additions. We modified our code to remove only 18 words and add 2. This caused our accuracy to drop slightly to 91%, showing that only removing words is the most effective strategy.

At this point we could not think of any plausible ideas to improve our results. For our final assignment submission, we reverted to using the algorithm from submission 4.