NLP coursework

Maxim Khovansky

November 2024

1 Preliminary sentiment analysis

1.1 Preprocessing

Initially, as the first combined preprocessing step, I tokenised the input text by splitting it by whitespace characters ("s+" in regex) into a list of strings (tokens) and performing basic data cleaning and normalisation: I split the tokens further by punctuation marks and converted all the tokens into their lowercase forms for normalisation.

1.2 Feature extraction

For feature extraction from a given text, I simply tokenised it and extracted each token as a feature. For weighting, I used the bag-of-words method: the weight assigned to a token was the number of times it appeared in the given text.

Additionally, every time I extracted previously unseen features from a text, I added them to the global feature dictionary to keep track of how many features my classifier is training on.

1.3 Cross-validation

For cross-validation of my classifier, I created a function that split the dataset into k parts (with k being a parameter of the function). If the size of the dataset was not divisible by k, I rounded up, thus leaving the last part a bit smaller than the rest. I then ran a for loop over each of these parts, whereby for a given iteration, one part functioned as the test set and all the others, combined, constituted the train set. Consequently, I trained the classifier on the train set and evaluated it on the test set using four metrics: precision, recall, f1-score (all computed using the 'weighted' method), and accuracy. Since the latter was not provided by the pre-imported SkLearn method, I had to compute the accuracy manually. I decided to add accuracy as a metric because the classes had a lot of instances - even though the negative class had fewer instances - making accuracy relatively reliable in this case.

1.4 Error analysis

The confusion matrix revealed that the number of false positives was greater than the number of false negatives, but the false negative rate was higher than the false positive rate. Both results were expected, since the positive class had almost twice as many instances, meaning both that had more training data for the class - improving performance - and that it would have more false positives all else being equal.

As part of preliminary error analysis, I also created a file containing all the false positive and false negative data, as well as their tokenised forms. From some of the examples - such as "Sorry, the blacks I know are actually great, successful people.They aren't taken in by Dems BS. They are very smart, educated!" - which the classifier labelled as negative, it was clear that the model did not fully learn the sentiment of individual words. In the given example, the number of positive words ("great, successful, smart, educated") was greater than the number of negative words ("sorry", "BS"), yet the model misclassified it. I took this into account when improving on the preliminary model in question 5.

2 Improved sentiment analysis

I tried a great variety of different methods and techniques. At first, to address the problem identified in question 4, I created two additional features for each datapoint: from sentiwordnet, I imported an opinion lexicon; I then counted the sentiment of each of the token words in the datapoint, and computed the average positivity and negativity scores of the text. This allowed the model to have direct access to the lexical emotional loading of the text, helping it avoid the mistake descrbed in section 4, and immediately improved the cv (cross-validated) accuracy from 85.1 (to 1 d.p.) to 85.5.

For feature weighting, I implemented tf-idf, which in addition of taking into account how prominent each token was in a text, also took into account how informative it was across the entire dataset. This weighting method gave me the most significant performance boost of all the tried methods, boosting cv accuracy to 86.4.

For text preprocessing, I implemented a lemmatiser, which would normalise words to their dictionary forms, or leave the tokens as they were if they did not have a known dictionary form. This normalisation method also boosted cv accuracy to 86.6.

Another big boost came from adding bigrams - unique pairs of tokens - to the feature vector of each datapoint. Since auxiliary words such as "don't" can totally invert the meaning of the subsequent word, this gave my model a lot of useful information, improving cv accuracy to 87.0.

The final significant performance boost came from changing the class weighting inside the LinearSVC classifier to 'balanced' - since my data was originally imbalanced, as described in section 4 - this reduced the bias of the model towards the dominant class and therefore improved cv accuracy to 87.5.

The ultimate performance gain to nudge the accuracy to 87.5 came from splitting all the emojis in the data to be recognised as different tokens: naturally, since emojis aren't agglutinative like letters are, individual emojis are more informative than sequences of emojis; hence, the performance boost was expected.