ECS763 Natural Language Processing Coursework 1: Sentiment Analysis from tweets

Aditya Gupta (Student Number: 240754763)

October 2024

Overview

The solutions to tasks 1-4 are in file "NLP_Assignment_1.ipynb" and for task 5 in the files "NLP_Assignment_1_5.ipynb" and "Final_Results.xlsx". This report discusses my work and findings while working on these tasks.

Question 1: Simple Data Input and Preprocessing

Implementing `parse_data_line()`

This function helps extract the text and label for each line in the input file. The input file has the headers, 'ids', 'text' and 'labels'; the line passed to this function contains the entries for each category. As we are only interested in the text and labels we extract those and return them such that they can be added to the raw_data list.

Implementing `pre_process()`

The code for this part is directly taken from the pre-processing step defined in Lab 1. This involves: 1) Using regular expression operations to Separate the punctuation at the beginning and end of a word; 2) Lower-casing the text to normalize it; 3) Tokenising the text by splitting it on trailing spaces.

Question 2: Simple Feature Extraction

The feature extraction step uses a bag of words implementation, that is, here each token's frequency serves as its weight. Here we have a local and global feature dictionary with the global dictionary keeping track of frequencies of tokens across all text samples or documents.

Question 3: Cross-validation on training data

In K-fold cross-validation, we want to compare the results of training and testing the model on different divisions of the same data to see how well the model generalizes and how well it works on unseen data. This is done by training and testing the model K times, where each time testing is performed on a different 'fold' while training is done on the k-1 remaining 'folds' of data. The implementation begins by doing this split, followed by training the model and predicting the associated labels for the current fold. Here we separate the inputs and labels from the test data as the inputs are used as a parameter for the 'predict_labels' function, while the test labels are used as an input to the 'precision_recall_fscore_support' and 'accuracy_score' function which are used to compute the performance metrics. These are subsequently stored in the cv_results variable, which is then appended to the final results which are returned by the function.

Question 4: Error analysis

To perform the error analysis I stored the original text, features, predicted and true labels for all false positive/negative cases in a separate file ("false_positives_and_negatives.tsv"). My key findings were:

1. **False Positives**: Many of the misclassified examples had ambiguous, neutral or sarcastic tones. A good example of this is: "@Starbucks your employees at 6th and 14th in NYC need an attitude adjustment, going across the street to Dunkin where they won't be rude" was potentially predicted as positive due to the inclusion of brand names (if the model considers them positive words) and a lack of direct negative markers.

2. **False Negatives**: Many positive examples were marked as negative, potentially due to use of neutral language and no strong sentiment as part of the text. For example: "davetaylor2112 Hey there David Taylor. Nice to have you along sir. How is your Friday? #mushroomfm".

Just from analyzing the false negatives/positives file, it can be deduced that the model seems to struggle with cases which don't have strong language. Also, ambiguous, polite or neutral language confuses the model due to a lack of nuanced context. This suggests a need for feature extraction methods that capture sentiment phrases better, such as n-grams as they can capture phrases like "not good", or "very bad". Also, many texts contain URLs, user tags and hashtags, which add noise to the text samples. URLs don't contain any clear sentiment markers and @ and # signs with text make it hard for the model to tell apart the same words as it doesn't understand that '@hello', '#hello' or just 'hello' have the same word. By analyzing the precision and recall values, the model seems to perform better at classifying negative cases than positive ones. There were a lot more False positives for the Positive Class, which makes sense as the dataset is imbalanced and is heavily skewed towards positive examples with around two thirds of all text samples with ground truth label as positive.

Question 5: Optimizing pre-processing and feature extraction

Based on the error analysis it is abundantly clear that, using Bag of Words alone is not a sufficient model for feature extraction as it struggles to capture the context of more ambiguous and neutral texts. Potential Fixes for this are: 1) N-grams; 2) TF-IDF feature extraction. Adoption of Advanced preprocessing methods is also necessary to reduce noise within the text contributed by hash-tags, user-tags and URLs. To address this I incorporated pre-processing and feature extraction methods such as URL Removal, Punctuation Separation/Removal, User&Hash tag removal, Applying Porter-Stemmer and Lemmatization, Stop word removal and generating N-grams. I conducted a total of 26 experiments of which these were the key findings:

Experiment	Configuration	Accuracy
0	Presence/Absence of Word + Punctuation Separation	0.8562
1	Bag of words + Punctuation Separation	0.8580
8	Bag of words + Bigrams + Punctuation Removal	0.8722
11	Bag of words + Trigrams + Punctuation Removal + Lemmatization	0.8718
13	Bag of words + Bigrams + Punctuation Separation + Lemmatization	0.8692
17	Bag of words + Quadgrams + Punctuation Separation	0.8703
19	Bag of words + Bigrams + Punctuation Removal + Tag Removal + Lemmatization	0.8785
20	Bag of words + Bigrams + Trigrams + Quadgrams + URL Removal + Punctuation Removal + Lemmatization + Stopword Removal	0.8417
21	Bag of words + Bigrams + Punctuation Removal + Lemmatization + Stopword Removal	0.8696
24	TFIDF + Bigrams + Punctuation Removal + Tag Removal + Lemmatization	0.8662

Despite high expectations for **TF-IDF**, its configurations underperformed compared to Bag of Words. A potential reason for this could be the dataset being skewed towards positive examples. Frequent words can be important indicators of a dominant class, and down-weighting such words can reduce accuracy, which can be true when using TF-IDF. TF-IDF paired with bigrams reached 86.62%, which was slightly better than the baseline case(experiment 1 in the table). While BoW with Bigrams gave us some of the highest-performing cases. The **Best Performance Configuration** was *'Bag of Words + Bigrams + Punctuation Removal + Tag Removal + Lemmatization'* which achieved the highest accuracy at **87.85%**. Experiments also show that adding too many n-grams leads to diminishing returns as they potentially start capturing noise again. Removing **User & Hash tags** also improved performance by making more generalized tokens. This reduced noise and made words accompanied by the @ or # symbols actually contribute to sentiment detection. I also tried adjusting LinearSVC's 'C' parameter and adding class_weight='balanced' which didn't improve the performance and sometimes even made it worse. Using Lemmatization was generally more accurate compared to stemming (as expected from what I learned in the lectures) and thus I ended up using the method in more experiments. The experiments proved that the improvements were incremental, with the addition of each method resulting in relatively modest gains. This reinforces what was mentioned in the tips document to only expect marginal or small improvements. This task showed the importance of strategic layering of different techniques. To further optimize this model one could consider adding stylistic features such as a count of the number of capitalized words or the number of punctuation marks as that could potentially capture a strong sentiment in the text and help with the overall tone detection.