Collin Paiz (1576109)

Professor Anastasiu

COEN 140

23 April 2023

Program 1

Text Classification

Rank & F1 Score

Rank: 1

F1 Score: 0.9529

Leaderboard: Visible

My Approach

My approach to text classification begins by first reading in the *train* and *test* datasets, and then separating the documents from the labels (in the *train* dataset) into two separate arrays. After that, I perform preprocessing techniques on the data to better prepare the data for classification. After extensive testing, I determined that the best combination of preprocessing techniques would be to make all words lowercase, remove punctuation, remove stop-words, and perform lemmatization. After preprocessing the data, I created a subset of the *train* dataset to run isolated tests in order to find the optimal values for parameters *c* and *k*. I tested different values for *c* and *k* by splitting the data in the *train* subset and performing classification on those documents and computing the F1 score relative to the actual values provided in the *train* subset. Whichever combination of *c* and *k* yields the highest F1 score is saved for classification on the *test* dataset. After that, I prepare the *train* and *test* documents by turning them into sparse matrices and normalizing the values (for cosine similarity during the classification step). The first step to creating these matrices involves using the *c* parameter to create a vector of *c-mers* for each document in both the *train* and *test* datasets. Then we take the *train* list of vectors and build the *idx* to map words to IDs. Next, we take both lists and build sparse matrices from each of them using the *idx* dictionary to ensure that both matrices exist in the same Euclidean space. And lastly, we normalize the matrices to prepare for cosine similarity. After building both matrices, we can begin running the classification predictions. To classify the *test* documents, we iterate through each vector in the *test* matrix and perform its cosine similarity with the *train* matrix. If no neighbors are found, we simply choose a random classification for the text by picking a number between 1-4. If neighbors were found (*train* documents that have similar words in them), we get those indices and count how many occurrences of each class there are in the neighbors.

The most frequent class is chosen as the *test* document's predicted label. If two classes are tied, we perform a weighted vote based on how similar they each are; whichever is more similar wins the tie and the *test* document is given that classification label. After running tests on the entire *test* matrix, we can take the list of classification labels and output them to a file.

My Methodology

The KNN approach I chose implements cosine similarity on sparse matrices; I chose this approach because of its efficiency and accuracy. Firstly, the use of sparse matrices drastically decreases the amount of memory needed to store the document representations in a matrix; because the documents are so short and the dictionary of words is so large, representing each of them as a vector results in a vast majority of zero-values. Using sparse matrices eliminates the need to store these zero-values, while maintaining the document's integrity in a vector format, relative to the Euclidean space. Secondly, I chose to use cosine similarity because it is an efficient and effective option for classifying many texts. Cosine similarity handles a high dimensional Euclidean space (large number of features/words) very well compared to alternative distance/similarity algorithms.

For my text preprocessing implementation, I chose to make all words lowercase, remove punctuation, remove stop-words, and lemmatize the documents. I chose this approach after performing extensive tests on a subset of the train dataset by splitting the data and trying different preprocessing combinations until I found which combination yielded the highest F1 score (2000 rows only). The best results I achieved with the subset was using the previously mentioned combination, which got an F1 score of 0.809166 (c=4, k=10). This score was shortly followed by using the same combination with stemming, which yielded an F1 score of 0.808019 (c=4, k=10), and then using only lemmatization, which yielded an F1 score of 0.803997 (c=6, k=10). Including stemming in any combination yielded a lower F1 score aside from performing no preprocessing at all.

To choose values for parameters *c* and *k,* I tested values between 1-10 for both parameters and determined which combination of *c* and *k* yielded the best F1 score. I performed this testing using the same train subset as previously mentioned. My program performs this step to find *c* and *k*, and then automatically uses those values for the actual test set.

Suggestions For Running

- Simply run all cells in order as is
- Keep stemming section commented-out (lowers F1 score based on *train* subset tests)
- Ensure *nltk* packages are downloaded beforehand
- *output.dat* may not record all rows, check *yayaya.dat*