Andrew Vattuone
Highest F1 Score: 0.9515
Leaderboard Ranking: 5th
April 20, 2025

Program 1 Report

The main goal of this project was to take a test document filled with text lines belonging to one of four types of news stories (labeled 1-4) and based on the text content, predict which label category each line should belong to. Using a training set with labeled lines, I was able to successfully accomplish this by training a model on the training data using cosine similarity and a k-NN comparison algorithm. However, it took me several iterations to arrive at my final result of an F1 score of 0.9515

For my first submission, I initially used the same preprocessing as I did in Lab 2. First, I needed to take each line in the training set and tokenize it so that it could be properly transformed into a numeric vector. I did this by removing punctuation using the Python standard string library, removing stopwords using the stopwords module from nltk.corpus, stemming and lemmatizing words using PorterStemmer and WordNetLemmatizer (respectively) from nltk.stem, and removing any empty strings. Afterward, I used a dictionary to store each unique word in the training set and gave it an index, and then used this to create a CSR matrix that stored the frequency of each word in each line of the document. Afterward, I normalized the matrix using its L-2 norm so that I could later perform cosine similarity on it to calculate the proximities for each row. For the test data, I used the same functions to tokenize and create a CSR matrix, although I used the same dictionary of unique words that I created for the training set instead of creating a new one. Afterward, I used my manual cosine similarity proximity calculation function that I developed in Lab 2 to calculate a list of proximities to each row in the training set for a single row in the test set (including the corresponding line of the dataset). To create my k-NN algorithm, I used calc_proximities on every row in test.dat to calculate its cosine similarity to all of the rows in train.dat, and then used a simple majority vote based on the labels of the train.dat lines with the top k proximities to predict the label of the current line of test.dat, and then wrote this value to predictions.dat. I decided to start by using k = 5, and for this submission I got a somewhat high F1 score of 0.9508. I did a second submission using the exact same code with k = 9, and I also slightly edited the way I got the top k proximities by just searching the whole list for the top k proximities rather than sorting the entire list (I did this since the runtime was very high), but got a slightly worse F1 score of 0.9495. At this point, I decided to try different preprocessing techniques to see if I could get better improvements in my results.

The first preprocessing technique I tried was using TF-IDF for the CSR matrix values rather than just raw frequencies. I implemented this by manually editing my build_matrix function to calculate TF-IDF(w, d) = TF(w,d)*log(N/(DF(w) + 1)) for each cell in the matrix, where w is the current word, d is the current document, TF(w,d) is how often w appears in d (term frequency), DF(w) is the number of documents in which w appears (document frequency), N is the total number of documents, and the + 1 is added to avoid division by 0. I needed to slightly edit the parameters of my build_matrix function to take in document frequency and inverse document frequency, as I used both of those values calculated with train.dat to process the data for test.dat. I also decided to use k = 7 for k-NN since it was between 5 and 9 and might've been slightly better than k = 5. However, I noticed a somewhat large decrease in performance, as my F1 score was 0.9471. I didn't think that the drop was too large to fix the TF-IDF algorithm specifically, so I tried adding other preprocessing techniques instead.

I wondered if I might get better results if I searched for short phrases rather than just individual words, as a short phrase a few words long might be more meaningful than multiple separate words. I searched for a library that could help me with this and found that I could already use ngrams from nltk.util to do this. I decided to just bigrams (2 word phrases) and trigrams (3 word phrases) since anything longer would take up too much space and be less meaningful. I also decided to remove any token that appeared in more than 90% of the lines of the training set. I decided to go back to k = 5 for k-NN since I didn't want to deal with as many changing preprocessing and k-NN techniques so I could better isolate changes. These changes gave me a slight F1 score improvement of 0.9489, although it was still lower than my first.

I realized that the problem might've been due to my TF-IDF algorithm, possibly due to the inverse document frequency giving less weight to words that appear frequently across all the documents even if all those terms appear in documents of the same class. I researched how I could improve this, and realized that I could offset the IDF problem by only multiplying TF by IDF if a term appeared in a document less than 10 times or in more than 50% of the rows. This would allow for important tokens that occur somewhat frequently (at least 10 times but in less than 50% of the rows) to not be as heavily penalized by IDF score. However, I was worried that this increase might be too large, so I did some more research and found that I could take the log of the raw TF score and add 1 to it so that the increase wasn't too extreme. Also, before building my matrix, I decided to calculate the discriminative weight of each word. I did this by finding the percentage of instances that a token appears in its most frequent label class out of all the total label classes (I called this max_ratio). If it was larger than 80%, I just set the discriminative weight to 1 since it's too common. If it was below, I set the discriminative weight equal to 1 + (max_ratio - threshold) * scale, where I set scale = 5. I then multiplied each TF-IDF score by this weight in build_matrix to increase the value of words that frequently appear in specific label classes. Finally, I slightly edited my k-NN algorithm to sum up all the proximities of the top k label classes and choose the label class with the largest summed proximity, as this would prevent more numerous label classes with low proximities from overriding less frequent but high proximity label classes. However, all of these changes just seemed to overprocess and dilute the data, as I got my lowest F1 score of 0.9355. I thought that the bigrams and trigrams might be a problem so I removed them (and changed calc_proximities to use sorting again for simplicity) but when I ran the code again without them I got a very low F1 score of 0.9137.

At this point I realized that all the weighting I'd been doing on the individual tokens was leading to too much data dilution. So instead, I decided to revert my code back to what it was for my first submission (just raw frequency counting). The only change I kept was the proximity summed k-NN I used in my previous two submissions, as I thought that this didn't overprocess the data too much. I then added back just the bigrams and trigrams, as those improved my F1 score earlier. When I submitted my predictions using this code, I got my highest F1 score of 0.9515. I wanted to submit another prediction file that only used bigrams and filtered out tokens that appeared in more than 90% of the rows or in less than 3 of the rows, but I ran out of submissions for the day so I just chose my 0.9515 submission as my final submission. I didn't end up ranking in the top 3, so none of my submissions ever had a rank. Note that my code might take a while to process since it uses many features.

Through Program 1, I learned a significant amount about k-NN algorithms and data preprocessing strategies. Although it could be used in some cases to use more advanced weighting of tokens (such as using TF-IDF rather than just raw frequency), changing token weights isn't always the best solution. If it's not done very carefully, then it can just end up diluting the data through making proximity distinctions less obvious. Doing more preprocessing of the tokens themselves (such as stemming, lemitization, and including bigrams and trigrams) usually has better improvements without as many potential detriments to accuracy. The only downside to certain forms of preprocessing is that it can lead to many more features being used (such as including bigrams and trigrams), leading to increased overall runtime. I also learned about the high importance of code runtime. Initially I manually calculated the dot products of my cosine similarity proximity, but I updated this to use numpy and csr_matrix functions, which ran much faster because these functions are implemented with lower-level languages like C that are much faster than Python. Finally, I learned that k-NN adjustments (such as choosing what value of k to use or whether to use simple majority voting or voting based on proximity sums) tend to have less drastic effects than preprocessing changes.