Michael Vanderlyn

HW 1 Amazon Reviews KNN Report

1. **User Name Registered on miner website:**
    a. ClassE
2. **Rank & Accuracy score for your submission (at the time of writing the report).**
    a. Rank: 33
    b. Accuracy: 74%
3. **Instruction on how to run your program.**

    Program is broken down into 5 subfiles in order to allow for modularity in design for the project, and to allow for more ease when pickling data structures at different stages of the program. 5 main programs are:

    - prepareFiles.py
    - preprocessFile.py
    - demensionReducer.py
    - preprocessTest.py
    - categorize.py

    Run each of these files sequentially and you will get the final ranking stored in test_ranks.txt. If you wish to only test a new test file, you can do this by modifying the global variable TEST_FILE in preprocessText.py, then run this file as well as categorize to get your answer in test_ranks.txt.
4. **Your Approach**
    a. Try 1

    My initial approach was to break down the different aspects of the project into different files. I was going to have one file preprocess the data, one file calculate weights, and one file run the KNN algorithm on the test data. For the first file I processed the input file line by line, stemming, and removing stop words. I then stored the words in a dictionary mapping the words to the number of times they occurred in different documents. After this I stored a list of dictionaries for every document containing the occurrence of each word in each document. I saved this information, then had a seperate file to calculate the weights of each term in each document, then normalize these values. By this point I was realizing just how much data 18000 documents is. I originally thought that the dictionary would be fast because of the O(1) lookup time, but with this many documents, and libraries made to assist matrices, I decided to switch over to matrices, and do more research into ways to reduce the amount of info I had to actually process.

    b. Try 2

    This time I had done a lot more research and decided I would first build the matrix into a 2d array using scipy's sparse matrices instead of a dictionary which allowed me to leverage libraries such as numpy, and sklearn to do preprocessing. Also the sparse matrix representation makes more sense because most of the dict entries would be 0 which wastes space anyways. I also realized I would have to do either feature selection, or dimension reduction, so I went with dimension reduction because I didn't want to have to deal with keeping track of the labels, and modify the test/train data based on the labels. Dimension reduction allows me to just store the transformation made to reduce the dimensions, and use the same transformation on the test data. Based on these

decisions, I broke my program down into separate files. First I preprocessed the training data, and calculated the weights at the same time. This takes a good amount of time, but is much faster than using a dictionary and having to code all the traversals manually. Then I run a dimensional reduction on the training data. I originally wanted to use PCA, but I learned that doing this on sparse matrices doesn't work well, so it is recommended to use a truncated singular value decomposition (SVD) instead. So I wound up using a truncated SVD to do my dimensionality reduction. I messed with the input and found that reducing the features down to 100 got me pretty good results. Finally instead of doing a linear search through every test document, and compare it with every trial document to find the closest K neighbors, I found a data structure which greatly aids with this. Sklearn has a representation of a k-d tree which allows you to query for nearest neighbors, so I used k-d trees to hold my trial data. I then got the nearest K neighbors, used their distances to weight their ranking, added up all the values, and based off this sum decided the ranking for the current test document. I messed around with the values for K and found K gets me pretty good results as well.

5. **Your methodology of choosing the approach and associated parameters.**
   In my research I saw a lot of cool information on running several tests with differing values for K and plotting them to determine the best value for K, and likewise doing this for inputs of dimension reduction functions. Given more time I would have liked to do more experimentation like this. However I just chose a random number to start out with and did a few trials changing the values for K, and how many dimensions I wanted to reduce to. Based on these trials I chose my parameter values.
6. **Efficiency of your algorithm in terms of run time. Did you do anything to improve the run time (e.g. dimensionality reduction)? If so, describe them and report run times with their respective accuracy before and after the improvement.**
   My program's categorizer runs in 82 seconds on average. The final section runs in O(N * k*n^(1-1/k)) where N is the number of documents in the test file, n is the number of documents in the trial file, and k is the number of nearest neighbors used in the KNN search. This basically boils down to the loop over all the test documents, and the lookup time of the K-D tree for each test document query.

   Originally when I was not doing dimensionality reduction my program would run in 5-6 minutes, and have a similar accuracy. The main things I could do to boost my accuracy would be to use a cosine similarity instead of a euclidean similarity. The K-D tree I used doesn't allow for cosine similarity to be used as the distance metric. Another thing I would do to increase accuracy would be a more thorough analysis of the values of K , and dimensions for the truncated SVD. Based off this analysis I could have a more informed choice for the best parameter values. I would also look into using feature selection instead of or as well as dimension reduction to see how that would impact runtime/accuracy.