Luis Ferrufino

G#00997076

4/17/20

CS 484-002

HW#4 Report

MINER TEAM NAME: Luis Ferrufino (worked by myself)

MASON USER: lferrufi

RANK: 5

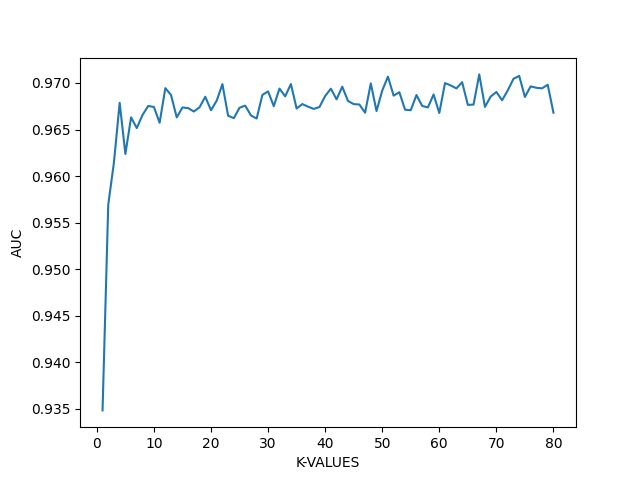
PUBLIC SCORE: 1.00

For HW4, I used the files found in HW4\_Ferrufino/src/ .

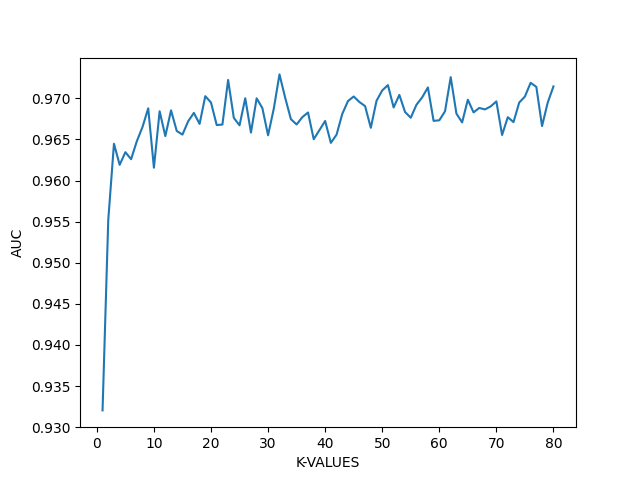
The first step was preprocessing (which was mostly in the form of feature extraction). This was done in src/extractFeatures.py . In this python script, for each mode, I load each sample, apply a Fast Fourier Transform and include it in a matrix; I saved the matrices of samples for each mode as src/baselineA.npy , src/baselineB.npy , etc., where each row is a vector containing the Fourier Transform coëfficients for each sample.

The second step was to create the distance matrix. This was done in src/makeDistanceMatrix.py . Here, I loaded each mode’s sample matrix and concatenate them. The distance matrix was computed by finding the distances between points and storing them; the result was saved as src/distMatrix.npy . In this same file, I also created an array which contained a mode label for each sample, saved as src/labelVector.npy .

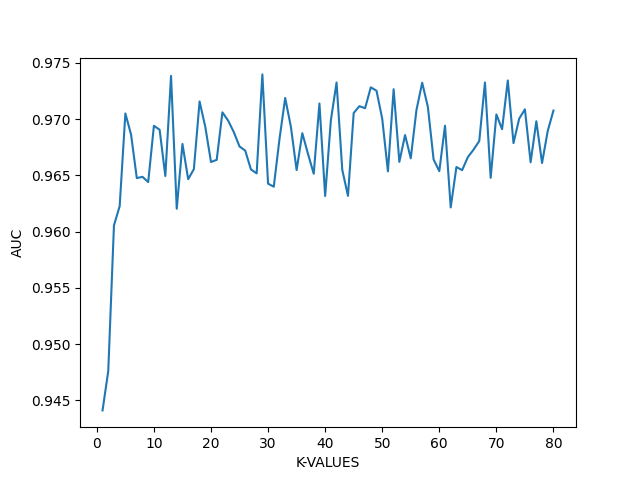
The third step was to calculate the LOF for each sample. This was done in src/lof2.py . Here, I randomly select some points to be in the training set, and others to be in the test set (most of the ugly index expressions are for recovering the original index or obtaining the kth nearest neighbour). I tried to make the sets even. Then I create a distance matrix for each set (using the original distance matrix). Then I create, for each set, what I call a neighbours matrix; an element on row I, column j is the index (on the original distance matrix) of the jth nearest neighbour for sample I; note that the 0th column will correspond to the sample itself, so in a sense it’s its 0th-nearest neighbour. Then, for each set, I calculate the LOF values. Next, I calculate the p-values. Finally, I figure out the corresponding ROC’s AUC (I used numpy’s trapezoid integration). I repeat this process for each k (from 1 to 80) for 100 trials; the average AUC is used to score each k. The results were, from worst to best, [ 1 2 3 5 7 11 28 24 14 6 23 27 8 60 80 47 17 49 20 55 54 38 35 16 25 15 58 18 39 10 68 37 31 9 57 26 65 66 46 45 36 4 44 21 71 42 75 19 69 33 40 52 56 13 29 59 53 70 30 50 72 41 32 63 78 12 77 43 76 62 79 22 34 48 61 64 73 51 74 67] . Thus, k=67 is the winner. This file also generates a corresponding graph after all k values have been tested:



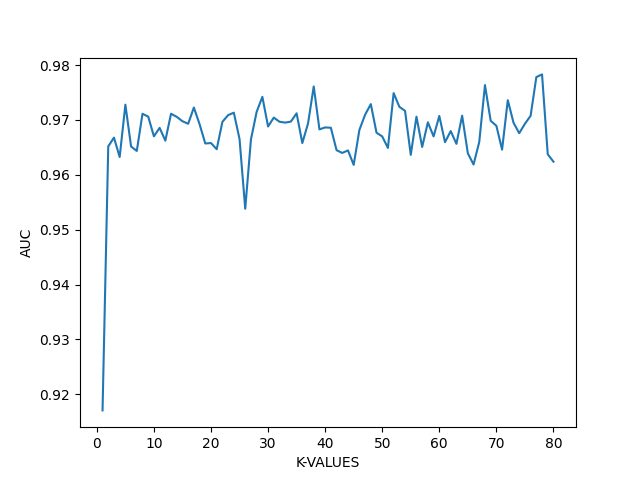
These are the results when 50 trials are used:



And when 20 trials are used:



When 10 trials are used:



Note how the fluctuations in AUC increase as the number of trials decrease. These images have been included in the Report/ folder.

The fourth and final step was to create the p-values for the test data and submit them to miner2. My file src/final.py is very similar to src/lof2.py , except it doesn’t do validation nor random sampling; this file has simpler index expressions as a result. This file creates the p-values and saves them as p-values.txt .

I had very low scores with my first submissions, but I think that was because I was using a very low number of trials (10) to obtain my k value to be used in src/final.py . When I switched to 100 trials, I obtained k=67, and got a 1.00 score.

Running src/lof2.py with 10 trials takes about 5 minutes. With 100 trials, it takes about 50 minutes. Running the rest of the files takes about 10 minutes also. So the whole procedure takes around 60 minutes in total.

I used the numpy, random, and matplotlib libraries to complete my homework.