Nearest Neighbor Lab Report

1

For this lab, we were assigned the task of implementing the K-Nearest-Neighbors (KNN) learning algorithm. With this method, the output class is determined by first determining the K training instances which are closest to the data to be classified. Then, combining the classification of the closest K training instances to form a prediction for the output of this data. There are many slight variants of this algorithm which either make it 1) applicable to more problems or 2) better at solving current problems. One example of a variant which allows the model to solve new problems is the regression variant. To use regression with this model, you use the same steps from earlier but instead of combining a 'class', you somehow combine the targets of the nearest neighbors. An example of a variation which allows better problem solving is the weight distribution. For some problems, it may provide better accuracy to give priority to training instances which are closer to the point, rather than equally weighting all K.

The first task was to actually implement the basic KNN algorithm. After completing the homework and reviewing the slides, I was fairly confident in my ability to get at least fairly close to the correct algorithm. One roadblock I ran into was the data representation. I once again made the mistake of using python lists instead of numpy arrays which made my code significantly less performant, but much easier to get working. For whatever reason, I had it in my mind that I did not want multiple arrays to represent the training instances. This led to me appending two elements to the arrays which represented the output class and the distance to the current training element. If I had gone for a different approach of storage, my code likely would have run much quicker and maybe ended up more readable. The only other hurdle to overcome was trying to share one base class which handled many different kinds of inputs/outputs. My class is able to handle many combinations of i/o types which makes it very versatile, but harder to code.

Once my code was written, I was able to very quickly get correct results for the debug set with both 'no_weights' and 'inverse_distance'. There was code from previous labs for decoding the arffs which I was able to recycle in this lab to help save me some time. While I am not particularly confident in my exact evaluation set accuracy, I am confident my model performs well and classifies it reasonably. My test set accuracy for the diabetes problem was 89.063%.

2

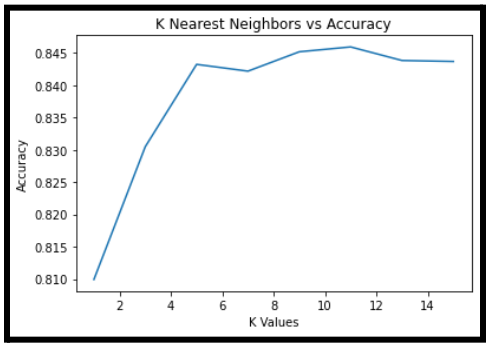
In this section, we were sent out to try our classifier on a few different problems with a few different setups. The main problem we experimented with was the 'magic_telescope' problem. This problem's datasets are each of considerable size which causes KNN to perform very poorly. KNN's prediction/scoring runtime is modeled by $M * N * Z + M * N * \log M$ which becomes $O(M * N * \max(Z, \log(M)))$ where M is the number of instances in the training set, N is the number of instances in the test set, and Z is the number of attributes in each element. This big O comes from the fact that for each instance you want to classify, you must find its euclidean distance to each other point. To find a point's euclidean distance from another, you must look at each attribute of each instance (Z) . Therefore, to find the points with a

minimum distance, we must look at each M and perform Z subtractions. On top of this, we need to order the M points so we can take the top K. This sorting process is done in O(M log M) time. Therefore, in cases where log M is larger than Z, the sorting process will 'outshine' the subtractions and replace the Z with a factor of log(M) This process must be done N times which gives us O(M*N* max(Z, log(M))). If we think of M,N, and max(Z, log(M)) as the dimensions of a cube, the volume of that cube gives the number of operations we have to do.

In the case of magic_telescope, M is ~12,000, N is ~6,500, Z is 10 and log M is 13. Therefore, the expected number of operations to predict/score the test set is some positive multiple of 12000 * 6500 * 13 which is just over a billion operations. Even if I had used numpy arrays and calculated distance in just O(M) time, the CPU level vectorization would not have saved me from the long sort times every iteration.

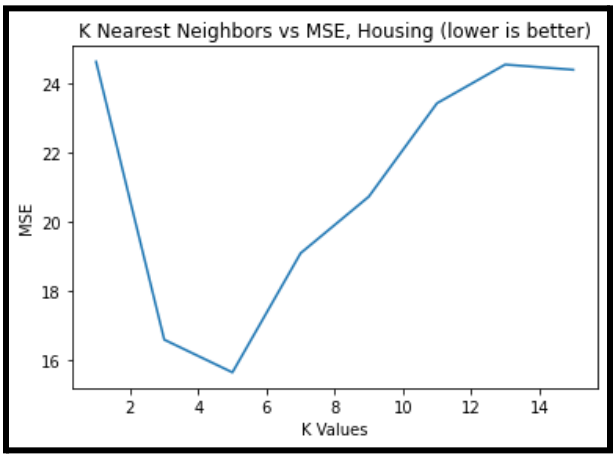
Another thing I focused on in this section was the difference normalization can make in your data. Without normalization, I was getting around 80% accuracy, but when I switched to normalized inputs, my classification accuracy jumped up to around 83%. This 3% increase in accuracy was quite impressive for a simple linear pre-process. I believe normalization is particularly helpful for KNN because the model has no way of adjusting weights to automatically partially normalize the data. For example, an MLP can tune down weights of a particular feature to be extremely small to normalize data into a reasonable range (though it can take a long time for large inputs). KNN however, can only 'see' the euclidean distance between points. When one feature goes from 0 to a billion, it will always outshine a feature which naturally ranges from 0-1. Therefore, we need to manually correct the 'overweighting' of certain features by forcing them into the appropriate range.

After experimenting with normalization, I tried using odd numbers from 1-15 as K values for KNN. Below is the graph of the accuracy values I got for various values of K. Accuracy increased significantly between 1-5 neighbors, but after that it mostly flattened out. While my accuracy was technically highest at 11, I believe differences between K=5 and K=15 may be more noise than actual improvement. I believe some of the reason accuracy stayed high when K got larger was due to the immense size of the training data. If there had been less training data, I believe different output classes may have started to sneak in and misclassify more edge cases, but when there is so much data, it is plausible that there will be 13 correct instances near you. This is especially true when you consider that we were not using distance weighting for these experiments.



3

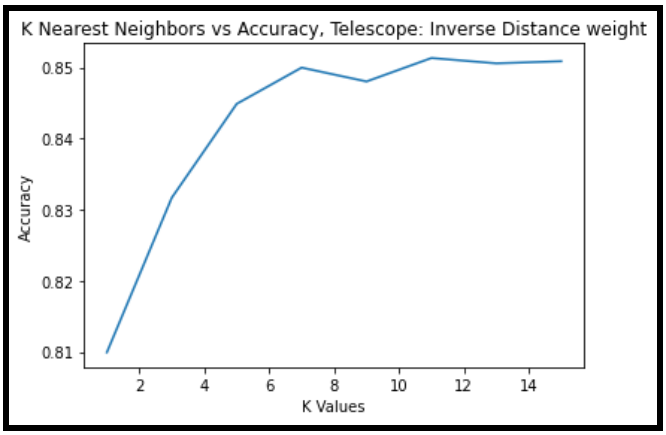
For section 3, we had to flesh out our regression algorithm and apply it to the housing price prediction problem. While I had anticipated regression problems from the beginning, there was some amount of polishing I had to do with my code to ensure it was functioning correctly. The only thing that really needed an overhaul was the scoring and prediction functions. These had to be modified to check if it was performing regression, then do some modification to their standard process. The code changes were fairly easy and I did not encounter any major issues.



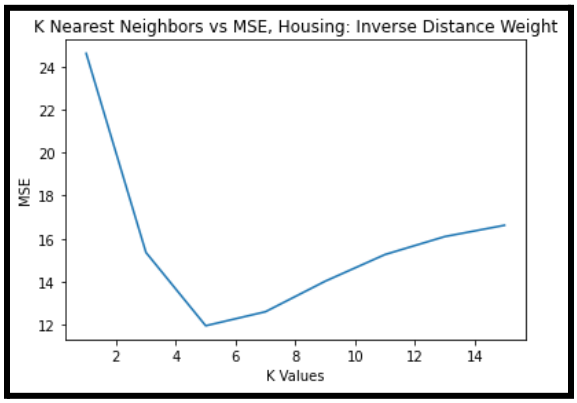
This dataset was significantly smaller than telescope, which made my life much easier when it came to running multiple slightly different versions of my model. Instead of doing a billion operations, I only had to do approximately 51 * 455 * 13, or about 300,000. After testing various K values, this is the graph I generated. This dataset has significantly fewer training instances which I believe led to larger K values giving poor accuracy. When more and more neighbors were considered, there were points less and less similar being equally weighted in the final prediction.

4

In this section of the lab, we did the same experiment as section 2.3 but with inverse distance weighting. The first time I ran through telescope 8 times, it took around an hour and a half to gather all the data. Rather than wait all that time again, I decided to use the power of threading to speed up my code's execution. I wrote some quick threading code to harness my other available threads and pretty soon I was off to the races. With my new threading code, the process only took around 20-25 minutes to go through all the data, quite a good speed increase! Overall accuracy was improved slightly. This algorithm managed to get 85% accuracy three times while no_weights was hard stuck at 84% across 3 different k values. Below is the graph relating different values of K to accuracy for the inverse distance weighting scheme.



After finishing up on telescope, I ran the same experiment on the housing data. Just like for telescope, I found significant improvement for accuracy. Starting at K=5, the inverse weighting algorithm dropped the MSE down by about 2, a ~10% reduction. However, an even larger difference was found in K values greater than 5. For values 7-14, the MSE still got worse, but it was always better than the naive no_weights version. Note the difference in MSE near the tail end of the K values.



5

In this section I simply had to apply my algorithm to yet another problem, credit-approval. This problem is somewhat different from other problems because it has continuous, nominal, and don't know attribute values. To handle nominal values, I simply checked if the nominal labels were the same and assigned either a distance of 1 or 0 accordingly. To deal with 'don't knows', I would always check if either value was a '?'. If either value was, the distance would always be .5. I figure this approach to don't knows is effective because it just uses the presumed average distance of .5. If I had instead chosen to use 1, that would be telling the algorithm that this attribute is as far away as possible which is more a worst case scenario than a good average. I believe using 1 leads to underrepresenting instances which have any don't knows when it comes time to predict. I tried a few k values and found 7 to be a number which provided good accuracy. My accuracy for this dataset varied run to run because of my random shuffling, but I averaged around 80-85% from my 20 runs.

6

This section is always great because it shows me how poorly optimized my code actually is. Scikit's classifier is able to rip through the telescope and the housing problems in under a second. For comparison, my code takes around 8 minutes to do the same task. I am unsure what kinds of magic they are using to get such a speed up, but I am pretty confident they aren't writing their for loops in python. Once I was done being amazed, I tried various hyper parameters such as different n_neighbors and weights and ended up getting results very comparable to my own hand rolled classifier/regressor. One interesting parameter I tried was 'algorithm' which allows you to 'de-optimize' their code to be running something more similar to my code. When I set 'algorithm' to 'brute' I was happy to see their code was taking 2.2 second instead of .7, making my code only 181x slower instead of 571x.

7

For this section, I used 'leave one out' to identify and use only instances which improved test set classification performance. I started out with 455 instances and .004 MSE, and was able to get that down to 119 instances and .001 MSE. 'Leave one out' can be very effective at reducing both noise and run speed, but you have to be very careful that you are not inducing overfit in your model. If you keep only instances which improve test set performance, you may very likely be throwing out all generalizability outside the test set. This approach feels like removing all parts of a car which do not make it go faster. By removing all the extra seats, all the car's mirrors, and the air conditioning, you are making your car really good at going fast around a race track, but it loses the ability to solve the more general car problem (going to the store, driving friends around etc..). While many points can be removed in KNN while maintaining accuracy, this approach felt like it was going too far in overfitting the problem. One solution to the overfit problem may be to introduce a validation set, but for the housing problem, there already isn't a lot of data to work with. This means shaving off a sizable validation set may cause us to lose significant accuracy on the test set.