K-Nearest Neighbors

Homework 1

Algorithm Description

Data

I created a custom data type using numpy wrappers to store the feature vectors and labels. This allowed me to have a vector, instead of a matrix, of the feature vector and labels. To access all the feature data I used trainExamples['data']. The only disadvantage of this method was I had to remember to decouple the data from the labels, and use the numpy squeeze() function when passing the feature data to built in numpy functions.

Basic KNN Algorithm

Numpy comes with a function called argsort() which returns the indices of a sorted array in ascending order. This function alleviates the need for bookkeeping (i.e., keeping track of labels) when sorting/querying an array. Numpy also has a function cdist() which calculates distances between two sets of n-dimensional data.

My KNN algorithm computes the distance between test data and the training data, sorts each matrix with argsort(), then returns the label that occurs the most in the first k elements of the sorted matrix for each testing example. The result is a list of predictions, one per test test example.

KDTree Algorithm

I used the built in library from scipy to construct and query the KDTree. I tried various options, but this was the slowest KNN algorithm. Once a KDTree is constructed, the tree can be queried and returns indices of the k nearest neighbors from the original dataset. In this case, it is just one index referring to the original data set. That index is used to retrieve the label and make the class prediction.

Condensing Algorithm

I computed the distance matrix for all training sets, chose random data points from each class (so there was one per class), and looped through all remaining points until the set was full or all the training data was classified correctly. This set, instead of the training set, was passed to the knn() function for classification.

Experiment Running Times and Accuracy

Basic KNN

By pre-computing the distance matrix from all the test data to the training data I was able to save some time. I still had to sort each random sample before calculating the k nearest neighbors. In general, running times increased with the number of neighbors and the number of samples used. Performance increased by using more neighbors and more samples. The most dramatic increase (20%s to 70%s) happened when the sample size went from 100 to 1000. From there, it increased slowly and topped out around 95%.

Pre-computation times:
Distance matrix (15000x5000) 2509.99999046ms
Sort time for sample size 100: 46ms
Sort time for sample size 1000: 733ms
Sort time for sample size 2000: 1482ms
Sort time for sample size 5000: 4134ms
Sort time for sample size 10000: 8299ms
Sort time for sample size 15000: 13245ms

ote: I really should have used quickselect, but it's not implemented in numpy or scipy. Sorting gets the job done though.

KDTree

This seemed to take longer that it should have for being a binary decision tree. The accuracy was good.

1-NN Condensed Algorithm

Aside from the included overhead from the distance matrix computation and sorting the samples, the running time was comparable to the 1 nearest neighbor algorithm.

Experiment Results

Confusion matrices/times in expdata.txt. Graphic confusion matrices in results/ folder.