Introduction

kNN is one of the algorithms used in machine learning for categorizing inputs based on the k closest data points in the training data. The brute-force serial implementation of the algorithm works as follows.

For each input

For each training data point

Calculate Euclidean distance from input to training data point

Update list of nearest neighbours

predict most common category for the list of nearest neighbours

This algorithm doesn’t scale well to large datasets thus it is necessary to parallelize it so it can run on multiple threads and/or computers to reduce the time it takes to run.

Parallelizing the algorithm to multiple threads should be easy using JOMP (A java implementation of openMP) as the for each input loop is embarrassingly parallel and we can use a jomp parallel for comment to parallelize it. The JOMP compiler can then be used to expand the comment and for loop into the necessary calls to the JOMP runtime library. However, the JOMP compiler is only compatible with java 1.2 syntax forcing me to use the runtime library directly as I had used generics and lambdas elsewhere.

Although the parallel version using JOMP works well on a single computer it can’t scale to multiple computers. Furthermore, the approach of parallelizing the outer for loop over input won’t work well for multiple computers as it requires duplicating the training data to each computer and for problems large enough to use multiple computers this might not fit into memory. To prevent duplicating the data, the data can be split between the computers, the local k nearest neighbours found on each computer and then the local nearest neighbours are combined. This approve doesn’t scale perfectly with the number of computers as the time taken to combine the local k nearest neighbours is logarithmic with the number of computers. Running this algorithm on multiple computers was straight forward using MPJ (An MPI implementation for java). however, I wasn’t able to also use JOMP to parallelize the algorithm within computers as the option multithreaded support wasn’t present in my version of MPJ. This lead to more local k nearest neighbours being found and combined than otherwise necessary.

The where other frameworks that I could have used to parallelize kNN including Hadoop map-reduce and OpenCL. I didn’t use Hadoop as it would make it harder to latter add support for using data-structures such as k-d trees and ball-trees to speed up finding the local k nearest neighbours. I didn’t use OpenCL as it didn’t appear to offer any advantage over JOMP when using a CPU and my algorithm was unlikely suited for running on a GPU.

Accuracy

To test that the parallel implementations of kNN gave the right answers, I tested them with different values of k and compared to result to the single threaded version. The data used for testing was various features of 150 iris plants of three different types. 15 of the data points where used as the test set (5 from each type of iris) and the rest as training data. The graph of number of neighbours versus accuracy is as follows.

Conclusion

kNN is easy to parallelize on a single computer. However, due to the size of some datasets it is necessary to use multiple computers. This can be achieved by splitting the data and calculating local k nearest neighbours and merging them however, it has less than linear scalability.