School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4040A Assignment 1 – Report

Tamlin Love
1438243
BSc Hons

March 10, 2019

1 Introduction

The k nearest neighbour (KNN) algorithm is a simple, widely used algorithm in used for classification and regression problems[1]. Its applications vary from detecting intrusive programs[5] to text classification [4] to the analysis of nuclear magnetic resonance spectra [3].
The algorithm itself is very simple. Consider a set P of m reference points $p_i \in \mathbb{R}^d$ and a set Q of n query points $q_j \in \mathbb{R}^d$. The aim of the algorithm is to find the k nearest (according to some distance measure) points in P for each $q_j \in Q$, for some integer k. In the brute force approach, the algorithm proceeds as in 1 in Appendix A.
In essence, we first compute the distance between each query point and each reference point and store it in a distance matrix of size $n \times m$. We then sort each row in the matrix before returning the $n \times k$ (keeping track of any swaps via the index matrix), before returning the $n \times k$ index sub-matrix containing the indices of the k-nearest neighbours to each query point.
Special attention must be given to the distance measure and sorting functions applied above. In this report, we consider the Euclidean distance measure (Algorithm 2) and the Manhattan distance measure (Algorithm 3), and for sorting we consider the quicksort (Algorithm 4), bubblesort (Algorithm 5) and mergesort (Algorithm 6) algorithms.
Because any distance measure in $\mathbb{R}^d$ must take $\mathcal{O}(d)$ time to compute, algorithm 1 above must take $\mathcal{O}(mnd)$ time to compute distances. Since quicksort and mergesort are both $\mathcal{O}(mlog(m))$ on average, and bubblesort is $\mathcal{O}(m^2)$ on average, the final algorithm must take $\mathcal{O}(nmlog(m))$ time to sort using quicksort or mergesort, and $\mathcal{O}(nm^2)$ time to sort using bubblesort. For large m, n and d, these complexities are problematic. However, as we shall see in Section 2, this can be significantly improved upon using parallel algorithms.

2 Methodology

As mentioned in Section 1, there are two computationally intensive sections in Algorithm 1: the distance computation and the sorting component. We shall apply Foster's Design Methodology to each of these sections individually in the hope of improving the performance of the algorithm.

2.1 Distance Computation

Consider the two distance measures (Algorithms 2) and 3). Because the only difference between these algorithms is the square of the difference in Algorithm 2 and the absolute value of the difference in Algorithm 3, they can be dealt with in the same way. No matter the algorithm used, the distance function is performed nm times in lines 4-6 in Algorithm 1. Furthermore, the computation of the distance between points q_i and p_j is in no way dependent on the computation of

distance between points q_s and p_t for $s \neq i$ and $t \neq j$. We will collapse the for-loops in Algorithm 1 into nm tasks which can be divided among processors.
In practice, this partitioning is done by the OpenMP **for** directive with the **collapse(2)** clause. The communication, agglomeration and mapping steps are performed by the OpenMP compiler.

2.2 Sorting

Consider the sorting algorithms (Algorithms 4, 5 and 6). These are all applied in lines 8-9 of Algorithm 1, which are repeated n times. Here, rather than focus on the loop in Algorithm 1, we will focus on applying Foster's Design Methodology to each sorting algorithm individually.

2.2.1 Quicksort

Here we note that each call to Quicksort recursively calls Quicksort on two portions of the list. We can therefore partition each recursive call into a separate task to be executed by a thread. This is done using the OpenMP **sections/section** construct. However, at a certain point, the overhead involved with scheduling threads becomes significant, so that sorting a small sublist in serial is actually faster than further partitioning the task. We therefore introduce a condition: if $high - low < c$ for some cut-off c, we will sort in serial. Otherwise, we will partition further. Communication, agglomeration and mapping is then handled by the OpenMP compiler.

2.2.2 Mergesort

Here we note that Mergesort, like Quicksort, recursively calls Mergesort on two portions of the list. Therefore, we can parallelisze it identically: by partitioning each recursive call into a separate task using the OpenMP **sections/section** construct. Once again, we employ the cut-off condition to reduce overhead.

2.2.3 Bubblesort

Here we employ an algorithm known as the Odd-Even sort or the Parallel-Neighbour sort [2], which essentially sorts pairs of numbers in the list in parallel. These pairs alternate from 0-1, 2-3, 4-5, etc. (even) to 1-2, 3-4, 5-6, etc. (odd). The algorithm, which can be thought of as the parallel version of the bubblesort, is listed as Algorithm 7 in Appendix A. In this algorithm we partition each inner for-loop (lines 4-7) into a task using the OpenMP **for** directive. Once again, communication, agglomeration and mapping are handled by the OpenMP compiler.

3 Empirical Analysis

We begin this section with a description of the hardware upon which the experiments below will be run.

4 Summary

References

[1] N. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[2] A. N. Habermann, "Parallel neighbor-sort (or the glory of the induction principle)," 1972.

[3] B. R. Kowalski and C. F. Bender, "K-nearest neighbor classification rule (pattern recognition) applied to nuclear magnetic resonance spectral interpretation," *Analytical Chemistry*, 1972.

[4] O.-W. Kwon and J.-H. Lee, "Text categorization based on k-nearest neighbor approach for web site classification," *Information Processing and Management*, 2003.

[5] Y. Liao and V. R. Vemuri, "Use of k-nearest neighbor classifier for intrusion detection," *Computers and Security*, 2002.

Appendices

A Algorithms

Algorithm 1 k Nearest Neighbour

```
1: procedure KNN(P,Q,k)
2:     distance ← [][]
3:     index ← [][]
4:     for $q_i \in Q$ do
5:         for $p_j \in P$ do
6:             distance[$i$][$j$] ← dist($q_i, p_j$)
7:             index[$i$][$j$] ← $j$
8:     for each $i$ in distance[$i$] do
9:         sort(distance[$i$],index[$i$])
10:    return index[$i$][$0 : k-1$]
```

Algorithm 2 Euclidean Distance

```
1: procedure DIST($q, p$)
2:     $distance \leftarrow 0$
3:     for $i \in [1,2,...,d]$ do
4:         $distance \leftarrow distance + (p_i - q_i)^2$
5:     return $\sqrt{distance}$
```

Algorithm 3 Manhattan Distance

```
1: procedure DIST($q, p$)
2:     $distance \leftarrow 0$
3:     for $i \in [1,2,...,d]$ do
4:         $distance \leftarrow distance + |p_i - q_i|$
5:     return $distance$
```

Algorithm 4 Quicksort

```
1: procedure QUICKSORT($index, distance, low, high$)
2:     if $low < high$ then
3:         $pivot \leftarrow$ Partition($index, distance, low, high$)
4:         Quicksort($index, dist, low, pivot - 1$)
5:         Quicksort($index, dist, pivot + 1, high$)
6: procedure PARTITION($index, distance, low, high$)
7:     $pivot \leftarrow distance[high]$
8:     $i \leftarrow low - 1$
9:     for $j \in [low, low + 1, ..., high - 1, high]$ do
10:        if $distance[j] \leq pivot$ then
11:            $i \leftarrow i + 1$
12:            Swap($distance[i], distance[j]$)
13:            Swap($index[i], index[j]$)
14:    Swap($distance[i + 1], distance[high]$)
15:    Swap($index[i + 1], index[high]$)
16:    return $i + 1$
```

Algorithm 5 Bubblesort

```
1: procedure BUBBLESORT($index, distance$)
2:     for $i \in [0, ..., m - 1]$ do
3:         for $j \in [0, ..., m - i - 1]$ do
4:             if $distance[j] > distance[j + 1]$ then
5:                 Swap($distance[j], distance[j + 1]$)
6:                 Swap($index[j], index[j + 1]$)
```

Algorithm 6 Mergesort

```
procedure MergesortParent($index, distance$)
    Create($index2$)
    Create($distance2$)
    Mergesort($index, index2, distance, distance2, 0, m$)
    Destroy($index2$)
    Destroy($distance2$)
procedure Mergesort($index, index2, distance, distance2, low, high$)
    if $low < high$ then
        $mid \leftarrow \frac{low+high}{2}$
        Mergesort($index, index2, distance, distance2, low, mid$)
        Mergesort($index, index2, distance, distance2, mid+1, high$)
        Merge($index, index2, distance, distance2, low, mid, high$)
procedure Merge($index, index2, distance, distance2, low, mid, high$)
    $l1 \leftarrow low$
    $l2 \leftarrow mid+1$
    for $i \leftarrow low; l1 \leq mid$ and $l2 \leq high; i \leftarrow i+1$ do
        if $distance[l1] \leq distance[l2]$ then
            $distance2[i] \leftarrow distance[l1]$
            $index2[i] \leftarrow index[l1]$
            $l1 \leftarrow l1+1$
        else
            $distance2[i] \leftarrow distance[l2]$
            $index2[i] \leftarrow index[l2]$
            $l2 \leftarrow l2+1$
    while $l1 \leq mid$ do
        $distance2[i] \leftarrow distance[l1]$
        $index2[i] \leftarrow index[l1]$
        $l1 \leftarrow l1+1$
        $i \leftarrow i+1$
    while $l2 \leq high$ do
        $distance2[i] \leftarrow distance[l2]$
        $index2[i] \leftarrow index[l2]$
        $l2 \leftarrow l2+1$
        $i \leftarrow i+1$
    $distance \leftarrow distance2$
    $index \leftarrow index2$
```

Algorithm 7 Odd-Even Sort

```
procedure OddEven($index, distance$)
    for $i \in [0, ..., m-1]$ do
        $j0 \leftarrow i\%2$
        for $j \in [j0, j0+2, ..., m-1]$ do
            if $distance[j] > distance[j+1]$ then
                Swap($distance[j], distance[j+1]$)
                Swap($index[j], index[j+1]$)
```