uOttawa

CSI 2110 [C] PROGRAMMING ASSIGNMENT 1

JOHN SURETTE
300307306
jsure011@uottawa.ca

K-Nearest Neighbours

K-Nearest Neighbours is an algorithmic concept that: given a set of data points (vectors, specifically) *S* we have some point *n* such that $n \notin S$ where we compare the *euclidean distance* of the nearest K points belonging to *S. KNN* typically has the complexity of O(Dn), where D is the dimension of the vector space S. In this assignment, we are analysing different implementations of a KNN using various types of PriorityQueue implementations using the given interface PriorityQueueIF. The following given classes for this assignment were:

LabelledPoint.java – Provided by Robert Laganiere
PointSet.java – Provided by Robert Laganiere
KNearestNeighbors.java – ChatGPT provided code via Robert Laganiere
PriorityQueueIF – Provided by Robert Laganiere; PriorityQueue Interface

Priority Queues

In all implementation of the KNN, the PriorityQueues were designed so that based on the value of the key (point.getkey() from LabelledPoint.java) the priority would be assigned based on the distance of the points closest to the query point, where the PriorityQueue is updated based on new points with the closest distance to the query point. The furthest points would be swapped out with the new, closer point.

PriorityQueue1 (PQ1.java) utilizes an ArrayList to track the nearest neighbours while iterating through the dataset file.
PriorityQueue 2 (PQ2.java) utilizes an array organized as a Heap – with parents and children sorting the array through comparators with the given key values.
Priority Queue 3 (PQ3.java) implements the java.util.PriorityQueue class.

Algorithmic Analysis

Each PriorityQueue implementation was tested for the average runtime (in milliseconds, or ms) for a file of 100 query points with a file of 1000000 base points for varying K values, as outlined in Tables 1-3:

Table 1: Runtime (ms) for varying K Values using PQ1.java

K Value	Trial 1 (ms)	Trial 2 (ms)	Trial 3 (ms)	Average (ms)
1	25669	34751	43107	34509
10	27271	53107	102900	61093
50	30751	86885	104473	74036
100	39350	85404	107372	77375
500	60800	106347	108477	91875
2000	106373	106373	100263	104336

Table 2: Runtime (ms) for varying K Values using PQ2.java

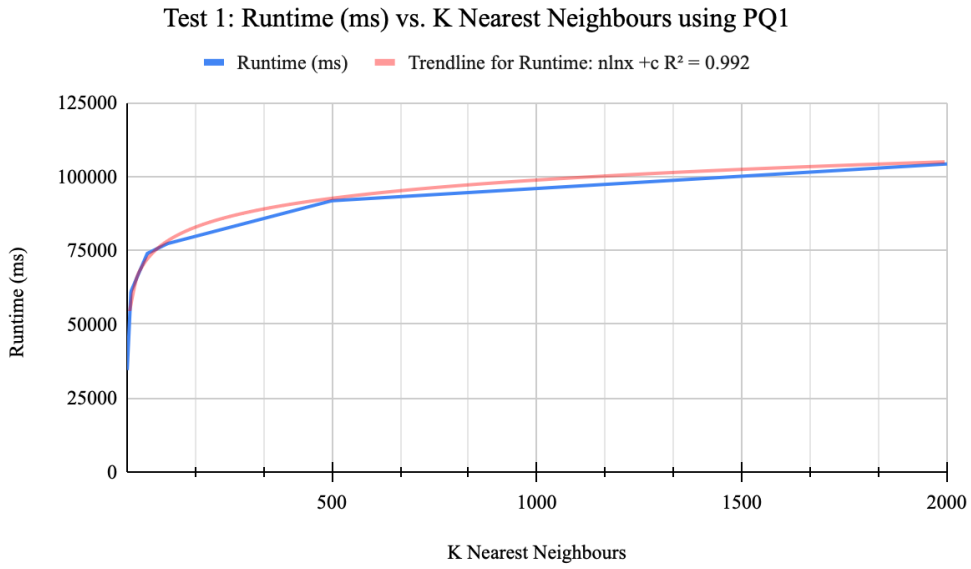
K Value	Trial 1 (ms)	Trial 2 (ms)	Trial 3 (ms)	Average (ms)
1	26899	29042	33703	29881
10	27271	29014	86885	47723
50	31382	32448	103973	55934
100	26511	68821	108096	67809
500	30424	85404	107207	74345
2000	111650	98995	108836	106494

Table 3: Runtime (ms) for varying K Values using PQ3.java

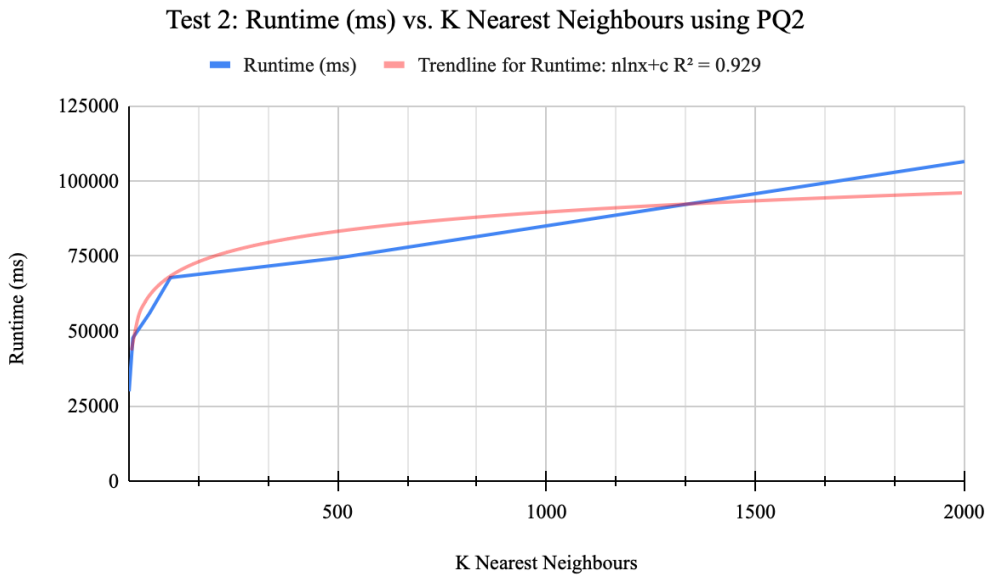
K Value	Trial 1 (ms)	Trial 2 (ms)	Trial 3 (ms)	Average (ms)
1	28927	64497	36912	43445
10	27271	45730	98091	57031
50	33337	56661	98364	62787
100	31733	60395	102378	64835
500	38249	84440	101103	74597
2000	79901	104384	105092	96459

The experiments were done in 3 Trials from K= 1,10,50,100,500 & 2000, from PQ1 to PQ2 and PQ3 in that order per trial. Notice that the runtimes are faster in trial 1 than trial 2, and trial 2 than trial 3. This is likely due to hardware/javaSVM issues rather than the code itself, as the

longer the testing period was, the longer the runtime. Determining the average amongst the 54 tests for each K value maintains a familiar runtime pattern, observed in Figures 1-3.

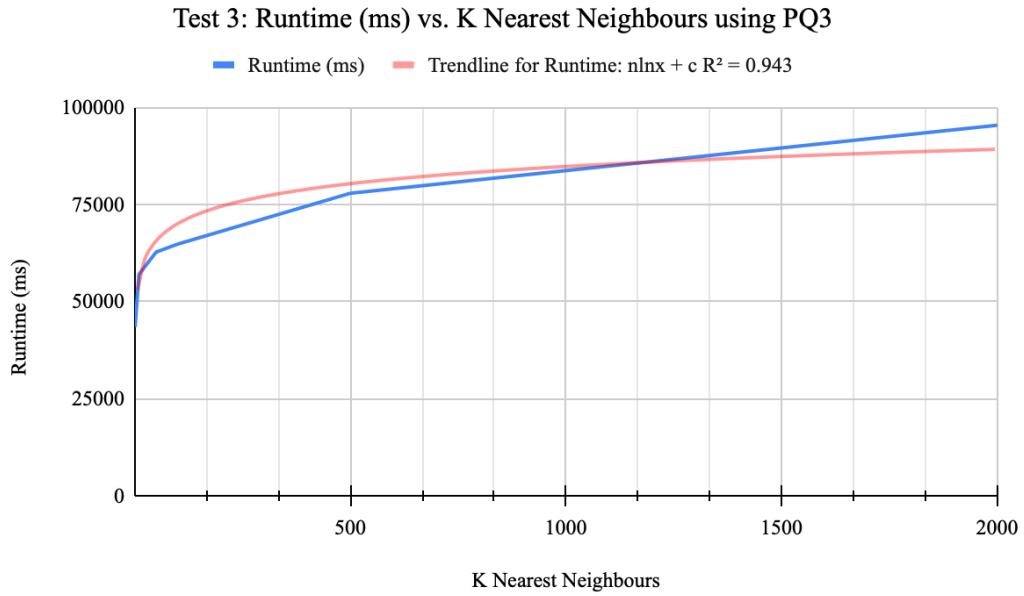


For the first Priority Queue (PQ) implementation, we observe a high correlation for a runtime complexity of the form O(nlogn), in this case nlnk + c, which is a pretty efficient algorithm considering the many iterations we have to accomplish for the algorithm to perform.



The second PQ implementation is quite fast with smaller dataset sizes, though with the 2000 point query for 100 query points has a slightly higher on average runtime than the other implementations. This is likely due to the rigorous upheaping that occurs every point called by

PQ2.offer(E element). The trendline for the runtime is akin to a $n\sqrt{n}$ function opposed to the *nlogn* form we observe in the other two PQs. This is also likely due to the heap implementation, which was made to organize an array as a heap rather than using a heap class itself. Overall, this implantation is the most efficient for point sets 500 or less from our experiments.



The third PQ implementation has the lowest upper bound average runtime. The runtime complexity has a high correlation with the function form O(nlogn). The performance is expected as this is implementing the java.util.PriorityQueue class, which is optimized for large datasets by Oracle and the official java.util developers. It beats both PQ1 and PQ2 in the long term.

Conclusions

Overall, we observe that the K values have varying effects on the runtime of each PQ implementation, where the value of K increases, the performance of the PQ3 KNN algorithm is the most efficient. The PQ2 KNN algorithm performs the best overall on lower values of K, showing the efficiency of a binary tree as a heap in smaller datasets. Given that sorted arrays will perform better than unsorted arrays by nature, it is sensible that the PQ2 implementation has better performance than the PQ1 KNN algorithm.