Fast serial KNN implementations

Amit Bhat
Microsoft
Hyderabad
bla bla bla
Email: amitgbhat@outlook.com

Kaushik Roy
Indiana University Bloomington
bla bla bla
Telephone
Email:kauroy@indiana.edu

***Abstract*—K-nearest neighbor is widely used data mining algorithm in many areas such as text, medicine. In this research, methods to speed up the K-nearest neighbor algorithm when executed sequentially have been proposed. All of these methods scan through the data set element by element. The first one uses a max heap implementation to speed up finding nearest neighbors of a data point. The second one uses a modified version of randomized selection to find the kth order statistic and then subsequently prune the data points to get the neighbors. The third method uses an insertion mechanism to find the appropriate position every time a data point is scanned in the data set in an initially maintained k-sized sorted array. Based on the size of the data set and the value of k, a suitable method is chosen from the above mentioned methods. Analysis of the methods for various data set sizes and k values has also been performed.**

***Keywords*—bla bla bla**

I. Introduction

The k-Nearest Neighbor algorithm is used in a variety of data mining applications such as data point classification, regression to name a few. The KNN algorithm finds the nearest neighbors of a data point from a set of labeled data points using a distance metric. It then resolves these nearest neighbor classes by a system of voting. The class with the most votes is assigned to the new data point. This is used in various applications, one such application being image processing where the pixel features distribution (color, depth, transparency, etc) of an image is given and the images in a database are most similar to the given image needs to be determined.

Another interesting application is regression. In cases where a mapping function from input to output cannot be computed or is computationally expensive, k nearest neighbors can be used to find the output value of the new input. Initially k nearest neighbors are computed for the new point, then the output values corresponding to these points are combined (averaged, calculating median, etc.) to determine an approximation of the output value. Then the error in the output measured and minimized.

Efforts have been made in recent research to speed up the time of execution of the k nearest neighbor finder method. [cite map reduce] Some of these methods include MapReduce implementations on a hadoop cluster and GPU optimization. But, very little research has advanced on how to improve the sequential performance of the nearest neighbor finding module. Preprocessing of the data can help speed up the process by pruning some of the input data set by a significant amount[cite LSH]. Some local search based methods making use of hill climbing and simulated annealing have been proposed. While these methods are fast on account of their local optimization, this also means that the results are an approximation and it is up to the user to decide if this trade off is acceptable [cite hill climbing, simulated annealing].

To sort through all the distances using the best sorting algorithm has time complexity of the order of O(m*logm) time where m is the size of the input data set as this is the sorting lower bound of comparison sorting. For a large input size, this execution time becomes unfeasible and therefore, research has advanced in exploring methods to better this time as mentioned

earlier. In this paper, a collection of methods have been proposed to achieve in many cases, linear time execution in the size of the input set. To the best of our knowledge, the methods proposed in this paper are unique and they can on combination with preprocessing methods, greatly improve the efficiency of the nearest neighbor finder. These methods can be used in conjunction with distributed architecture to achieve even faster efficiency. [cite MapReduce]

This paper is divided into 7 more sections. Section 2 provides some background on the concepts upon which the research is based, Section 3 explains the methodology adopted, Section 4 details the implementation of the proposed methods, Section 5 provides the theoretical time analysis, Section 6 shows the experimental results and graphs, Section 7 explains the inferences drawn from this research and section 8 discussed possible directions of improvement and concludes the paper.

II. Background

A. KNN Background

- In data mining, many times to analyze a new point from a data set relative to a set of candidate points whose behavior is already known (for example classification), it becomes expensive and complicated to consider all such candidate points in performing the analysis. Moreover, the behavior may not even be conditional on all of the candidate points. At such times some form of bucketing is required to know which of these points, which when considered, significantly contribute to the behavior of the new point. This problem presents itself as the ”nearest neighbor finder” problem or more specifically ”k nearest neighbors”.
- As already mentioned, it is used in classification, regression and many other applications in data mining.

- It works by using a distance metric. If all the points and their relations to other data points in a vector of data points can be modelled on n axes, in other words n dimensional data points that can be mapped on to a set of basis vectors allow for the use of distance metrics such as Euclidean distance, Manhattan distance to obtain a measurable degree of similarity between points. This notion of similarity can then be exploited to find the k nearest neighbors. But if the mathematical modelling of the data is not straightforward, other distance metrics need to be constructed that require domain specific knowledge. One example is if the values that a random function on the data can assume are binary, meaning that the data vector is a bit string, hamming distance can be used for finding nearest neighbors.

B. Heap Background

A heap is a binary tree that requires two additional invariants maintained at all times:

- The tree must be complete.
- For every node in the tree, the child nodes are less than or equal to the parent in the case of a max heap and the child nodes are greater than or equal to the parent in the case of a min heap.
- For this paper, a max heap has been used. Notice that the max heap invariants mean that the root of the tree is at all times, the largest element in the tree. The reason for choosing a max heap is that while maintaining a heap of the k nearest neighbors of a new data point, only if the distance of another point to the new data point is strictly less than the root of the tree, it makes sense to insert into the nearest neighbor heap. Why not use a priority queue? While it is true that the heap in concept, at least with respect to ordering of elements closely resembles a priority queue, heap operations are much faster. That is, discarding the root found at the root and then inserting the new element takes logarithmic time in size of the heap which is k. This is not true for a queue. The same operation on a queue will take linear time in the size of the queue. Therefore, linear in terms of k.

C. Selection algorithm background

- It is used to find the kth order statistic in a list of elements with an expected runtime of linear in the number of elements. The statistic required can be the kth smallest, largest, median, etc. It works by using the decrease and conquer design paradigm by partitioning the elements around a pivot and then continuing the search in either direction, as determined by the value of k and the element chosen as the pivot.
- There are two versions of the selection algorithm. A deterministic version in which the pivot is chosen using a deterministic approach called the ”median of medians” [cite deterministic paper] and a randomized version that simply selects the pivot element at random [cite randomized selection paper]. The randomized version has its theoretical premise in probability theory which in short explains mathematically that even though the worst case time complexity of the algorithm is quadratic in the size of the list, the list would have to be presented in a very specific way to achieve this run time. So, for a list of size, say 100000 elements, there are 100000 factorial orderings out of which a handful of cases pertain to the worst case time bound.
- Also, in practice owing to large hidden constants in the deterministic approach, the randomized selection works faster on average (actually a very large majority of the times).
- In this paper, k nearest neighbor has been implemented using the randomized selection as well as a modified version which considers the uniform distribution of the elements in the list and the nature of the k nearest neighbor problem to achieve a partition around a pivot that is closer to the start. This is explained in more detail in a later section.

D. Insertion sort background

- Insertion sort is a sorting technique that works by scanning each element in the input list and ”inserting” it in the appropriate position in the part of the list that is already sorted [cite insertion sort]. For each element, it needs to compare to at most all of the remaining elements in the list to sort the list. This means that in run time is quadratic in the size of the input list.
- When a sorted array is used to maintain k nearest elements of a large number of data points. Let the number of data points be m. So in the worst case, inserting into a k sized array for all of the m elements will take O(mk) time as opposed to O(mlogm) for sorting the distances and then picking the first k.
- In the KNN implementation using insertion sort in this paper, k elements are sorted using this technique and subsequent elements are ”inserted” into in the appropriate position in the k nearest neighbor list only if the subsequent element is strictly smaller than the last element in the list (which will be the largest as the list is sorted). This strategy is similar to the max heap implementation. Of course, as already mentioned the operations take more time theoretically than in the case of a heap.

III. Methodology

The proposed methods have been repeated for different data set sizes and values of k and a comparative analysis has been performed.

A. Max Heap method

The distances to the new point of all the points in the data set are calculated one point at a time. A k sized max heap is populated with the first five distances. Subsequent distances are inserted into the heap only if the distance in consideration is strictly less than the root of the max heap. After all the elements in the distances array are scanned and inserted into the heap as necessary, the final heap contains the k nearest

neighbors of the new point. Also, a reference to the data point with respect to which the distance is computed is also recorded during the insertion. This is important as we need to know what data points the k nearest neighbor distances in the heap correspond to.

B. Insertion Sort method

Here an array of size K is maintained. Initially k distances between the new point and points from the data set are populated into this array using insertion sort. So, a sorted array of first k distances which is obtained. This means that the last element is the highest element in the array. Now the rest of the dataset is processed one element at a time. Each distance between the element and the new point is compared with this last element (which is the largest element) and inserted into the array in the appropriate position only if it is strictly less than the last element. This means that the last element is discarded from the array. This technique is similar to the max heap implementation. Again, a reference to the point with respect to which the distance is computed is also recorded for reasons already mentioned.

C. Randomized Selection method

In this method, all distances to the new point from the points in the data set is partitioned around a randomly chosen pivot [cite randomized selection]. Here the goal is to find the kth smallest element. Next, if k is less than or equal to the index of the pivot, the search moves to the left of the pivot. Otherwise, it moves to the right of the pivot. If the search moves to the right the k value is adjusted to the partition index added to k. If the search moves to the left, the value of k is unchanged. This process is recursively repeated until the kth smallest distance is found. This follows directly from the implementation of the randomized selection algorithm. Then all the elements to the left of this element is returned as the k nearest neighbors. A key difference in this method is that the elements returned are not sorted.

D. Modified Randomized Selection method

This optimization is applied to the partition step of the randomized selection algorithm. In randomized selection implementation we choose a random pivot and balance the array around it. But in the scenario where k is very small when compared to the dataset size, it would be profitable if the pivot is biased towards the start so that the desired pivot (kth order statistic) is reached faster. To make this bias, a parameter is chosen (gamma) which can be dependent on dataset size and k and another parameter chi is chosen based on gamma. Initially before computing pivot the distance between start and end in the partition function is compared with chi. Only when the size of array to partition (distance between start and end) is greater than chi we apply an optimization. The optimization will be scanning through the array to partition from start to start + gamma to determine the minimum valued point. This point is now chosen as the pivot and rest of the process is carried out in the same way as the randomized selection. The intended purpose of deterministically determining pivot instead of randomized is to bias the pivot towards smaller values of the given array. Example, when chi = 7 * gamma, gamma = 2 *k, for every call to partition function the difference between start and end is compared with chi and if the difference is greater than chi, the minimum element from start and start + gamma is chosen as the pivot element, else the method chooses random pivot.

IV. Implementation

In this project C++ was used. This is because C++ being unmanaged code is faster than any managed code. The data points were generated using a random number generator. The distance function used for is the general l2 norm or more commonly known as the Euclidean distance. This is the distance between two points in a vector space. For example, in two dimensional vector space the distance between two vectors x1,y1 and x2,y2 in a plane is:

$$\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

. This can be easily generalized to n dimensional vector spaces as the difference between two In this paper, since time of the nearest neighbor calculation is of concern and not the time taken to compute the distance, two dimensional points have been generated for simplicity. The goal is to better the run time of the traditional approach which first sorts all the distances. This runs in time of the order O(mlogm) where m is the size of the data set. Therefore, four approaches have been proposed that achieve a better time bound. They are explained as follows:

A. Heap Implementation

- In this approach, a k sized max heap is maintained. 'k' being the number of nearest neighbors to find. In this heap the distance between every element in the data set and the data point with respect to which the nearest neighbors are being found is recorded. Along with the distance, a reference to the point in the data set that corresponded to this distance is also maintained as an index. So, if d is the distance between data point x and data point y, then when d is added to the heap, the index of y in the input data set I is also recorded. The heap is organized as a one dimensional array with the first element of the array being the root. The left child of a node which is indexed by x is at index 2x and the right child of the node is at index 2x+1.
- The first five distances from the input point is pushed into the heap as is, maintaining the max heap invariants. Subsequent distances are pushed into the heap only if the distance is strictly less than the root of the heap, or in this case the first element of the one dimensional representation of the heap. Through one pass of the entire data set, the max heap will contain the k nearest neighbors of the input point sorted by distance.

B. Insertion Sort implementation

- In this approach, we maintain a k sized array of sorted elements at all times during the pass through the dataset. For the input data point whose neighbors are to be found, the first k distances to the corresponding points in the data set are inserted into the array using insertion sort. Subsequent distances between the remaining m-k points and the input data point is

inserted into the array only if the distance is strictly less than the largest element or the last element in the sorted array. The indices corresponding to the distances in the data set is also maintained during this process.

- Therefore, just like in the heap implementation, through one pass of the entire data set, the k sized array will contain the k nearest neighbors of the input point sorted by distance. This approach is very similar to the heap implementation, the only difference being in the time taken for insertion of a new distance in the array. Naturally, insertion into the heap takes lesser time than insertion into a single dimensional array. However, this method is still faster than sorting all distances and picking the first k distances as the nearest elements.

C. Randomized Selection implementation

- In this approach, all the distances from the input point to every element in the data set is organized a m sized single dimensional array. Now, the randomized selection algorithm is used to select the ith order statistic from an array of elements. So, in this case 'i' is equivalent to 'k' and the kth smallest distance is found from the above mentioned single dimensional array. Now, the partition takes place in such a way that all the distances larger than the kth smallest distance is placed to the right of the kth smallest distance and all the distances smaller than the kth smallest distance is placed to the left of it. So, after finding the kth smallest distance, all elements to the left of it are returned as the k nearest neighbors of the input point.
- It is noteworthy to point out that in this approach the nearest neighbors returned are not sorted by distance.
- However, since this method returns the k nearest neighbors in one pass of the data set without any insertions and deletions, choosing k appropriately can ensure linear running time, i.e. linear in the size of the data set. This brings us to how can we ensure the choosing of a k that can partition the single dimensional distance array efficiently around the kth smallest distance so as to achieve most efficient runtime. This is explained in an optimization in the next approach used.

D. Modified Randomized Selection implementation

- Since k is very small generally relative to the size of the data set i.e relative to m, it is better to choose a partition index value closer to the start of the array as this can ensure that we reach desired pivot which is the kth smallest distance faster.
- For this reason, a parameter gamma has been introduced. This parameter provides a threshold based on the distance values as to how biased towards the start of the distance array does the partition index need to be in-order to ensure faster convergence on the kth smallest distance. For example, in a set of uniformly distributed distances of size 200, it is likely that most of the 20 smallest distances occur before the 50% mark which is below index 100. This can be attributed to the fact that if from this set a distance was chosen at random, it is as likely as any other distance chosen in another such independent random experiment. Once gamma has been chosen, the element with minimum value between the start index and the start plus gamma index is chosen to be the element around which the partition is carried out.
- Now, there is another problem. If the partition index chosen this way turns out too close to the start, then the bias in the positioning has to now move right instead of left. In the previous example of 200 distances, if the partition index chosen is 10, then it is likely that some nearest neighbors lie to the right of this index and not to the left. Therefore, in this regard, a new parameter chi has been added to account for this scenario. Thus, the difference between the start and the end of the array to be partitioned at each recursive step is compared to chi. If it is less than chi only then the gamma optimization of picking the minimum element between the start index and start plus gamma index is carried out. Otherwise, the partition index is chosen at random.
- So, in cases where the optimization is applicable, it out performs the randomized selection implementation, and when it is not applicable, it performs just the same. Therefore, on average the modified randomized selection implementation of the nearest neighbor finder performs better than the randomized selection without the optimization.

V. Time Analysis

In this section the theoretical run time of the various approaches is discussed.

A. Heap implementation

- The worst case run time of the heap implementation is:

$$O\left(m * log\left(k\right)\right)$$

. As mentioned before, in this method a max heap is maintained to store k smallest elements as and when elements are scanned from the data set. When a new element is scanned, it is initially compared with the root of the max heap (this contains the max element in the heap) and inserted into the heap if it is strictly less than the root. If the distances computed from the data set are in descending order, then the worst case scenario occurs when every element encountered needs to be inserted into the max heap as it is found to be less than the root. Insertion into the k sized heap takes

$$O\left(log\left(k\right)\right)$$

time. This means that the worst case run time of the max heap implementation is of the order:

$$O\left(k * log\left(k\right) + (m-k) * log\left(k\right)\right) = O\left(m * log\left(k\right)\right) \quad (1)$$

- Here k is the number of nearest neighbors required to be computed and m is the size of the data set. The first term in the expression (1) corresponds to initial k insertions. The second term corresponds to the rest of the data set for which distance is computed and then the distance is inserted into the heap. However, the likelihood of distances being in descending order is very low. If we assume that the data set is randomly scattered, then the distance from the given data point will be in random order. This would mean that most of the distances are not even inserted into the heap since when the newly computed distance is compared with the maximum element in the heap, it turns out to be greater than the max element in which case the data point is ignored. Hence, in practice the time taken for getting k nearest neighbors using a max heap is much less than the order of time mentioned in formula (1). But the order of growth of time with respect to change in values of k and m still follows formula (1).

B. Insertion Sort implementation

- The worst case run time of the insertion sort based implementation is:

$$O(m * k)$$

. In this method insertion sort is used to insert an element into a k sized array so that at all times the k-sized array is sorted. This also means that at any point of time the maximum element in the k sized array can be obtained in constant time (since it is the last element of the array as the array is sorted in ascending order). If the distances are sorted in descending order (when data set is scanned), the worst case run time scenario is encountered where every element is inserted into the k sized array. This would take at most k comparisons in order for the element to be inserted at the appropriate position in the array. This also means that the last element is discarded from the k sized array. The time complexity in the worst case is given by:

$$O(k * k) + O((m - k) * k) = O(m * k) \quad (2)$$

- The first term in the formula (2) corresponds to initial k insertions into the k sized array (where at most k comparisons are done). The second term corresponds to (m k) insertions as the dataset is scanned and the distance always turns out to be strictly less than the maximum element in the k sized array. Here the insertions are also assumed to have at most k comparisons to be successfully inserted into the k-sized array. However, similar to the max heap's case, the likelihood of distances being sorted in descending order is again very low. Hence, in practice the time taken to obtain the k nearest neighbors of an input point is much less than the time stated in formula (2).

C. Selection implementations

- The worst case run time of both these methods is

$$O\left(m^2\right)$$

. Since the method of selection of the element around which to partition the distances array in each recursive step is probabilistic, there is always the possibility when the element chosen in most of the recursive steps achieves a highly unbalanced split. For example, a 90-10 split meaning 90% of the element are placed to the left of the pivot and only 10 are placed to the right of the pivot or vice-versa. In such a scenario we encounter the worst case running time of the randomized selection algorithms.

- So, if the 2nd smallest element in an n sized array is to be found, and each partition reduces the size of the array by only one element. Then if each recursive step takes linear time, then the run time in this case is given by:

$$O(m) + O(m - 1) + O(m - 2) + ... + 1 = O\left(m^2\right) \quad (3)$$

- Each term in equation (3) corresponds to the time taken for the (m ith) recursive step. The likelihood of the distances being distributed in order to achieve this worst case scenario is extremely low. Therefore the average run time of the randomized selection algorithms are very much lower than the time projected in equation (3). The modified version with optimization reduces the time taken by some fraction of n on average which is still a constant times n and hence still has a theoretical best case run time equal to that of the version without the optimization. However, in many cases it performs better as will be clear from the experimental results section.

VI. Experimental Results

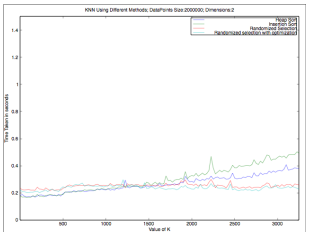

Fig. 1: Crossover point

Fig 1 shows the performance of the different implementations for small values of k and also shows at what value of k, the randomized selection methods start to out perform the heap and insertion sort methods.
The size of the data set used was 2 million 2 2-dimensional vectors and the point of crossover found to be around the 1500 mark.
Fig 2 shows that the selection methods greatly out perform the heap and insertion methods.
The size of the data set used to generate this result was 15 million 2-dimensional vectors.
Fig 3 shows a plot of the k value at which the selection methods start to out perform the heap and insertion methods against different data set sizes.

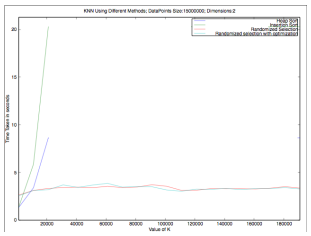

Fig. 2: Performance Comparison

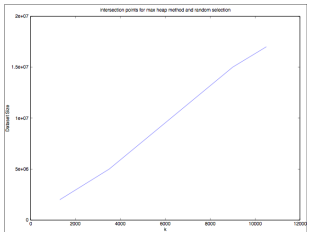

Fig. 3: plot of k vs data set size

VII. Inferences

A. Independence on k value

From Fig 1 and Fig 2, using two different data sets it is clear that randomized selection run time is independent of the value of k. This is in keeping with the theoretical analysis performed in the time analysis section.

B. Right method based on k value

Also, observed in Fig 1 and 2 is that for insertion sort and heap method of finding k nearest neighbors, it can be observed that the run time increases with increase in value of ”k”. This increase is greater in the case of insertion sort than in the case of max heap method. This aligns with the theoretical analysis performed on the algorithms which states that in the case of insertion sort the run time is linearly dependent on the size ”k” and in the case of max heap implementation, the time depends on logarithm of value ”k”.
However, for small values of ”k” with respect to the data set size ”m”, it can be observed that both insertion sort and max heap methods perform better than randomized selection. This can be attributed to the additional overhead required in case of randomized selection as explained in the time analysis section. Hence, the optimal method for getting k nearest neighbors can be decided from the value of ”k” with respect to the data set size ”m”. That is, for small values of ”k” max heap method can be chosen and for higher values of ”k” randomized selection based methods can be chosen. As mentioned earlier, in the latter case, the k nearest neighbors returned are not sorted based on the distance from the new point. An additional processing of a list of k items needs to be made if sorted items are required.
The decision between max heap implementation and insertion sort, for small values of ”k”, can be made based on the data structure available and complexity that can be tolerated. Given that max heap is relatively more complex than insertion sort, insertion sort can be used when complexity needs to be minimized without significant decrease in performance.

C. Relation between cross over point and m

From the Fig 3, it is observed that the value of ”k” at which the execution time of max heap method crosses (goes higher than) that of random selection is directly proportional to the data set size ”m”. The mean of the ratios of m:k from the experiments have been found to be about 1548.67, which is approximately the constant of proportionality. This can be used as an attribute to determine which of the methods to use for computing k nearest neighbors.

VIII. Conclusion and Future Work

Scalable implementations based on these methods, for distributed computing systems is a direction for further improvement on the achieved execution time. Also, probability models of the data set can be constructed to better determine parameters such as the ones used in the optimized random selection method.
In this research, fast serial implementations of the KNN problem and a novel optimized randomized selection method have been proposed. The efficacy of the methods have been proven and illustrated.

References

[1] B. Bla and B. B. Bla, *Some conference*, 3rd ed. Someplace, USA: Year, 2016.

[2] B. Bla and B. B. Bla, *Some conference*, 4rth ed. Someplace, USA: Year, 2016.