Approximate K Nearest Neighbors in High Dimensions

George Dahl
gdahl@cs.swarthmore.edu

Mary Wootters
mwootte1@cs.swarthmore.edu

Abstract

Given a set P of N points in a d-dimensional space, along with a query point q, it is often desirable to find k points of P that are with high probability close to q. This is the Approximate k-Nearest-Neighbors problem. We present two algorithms for AkNN. Both require $O(N^2d)$ preprocessing time. The first algorithm has a query time cost that is $O(d+\log N)$, while the second has a query time cost that is $O(d)$. Both algorithms create an undirected graph on the points of P by adding edges to a linked list storing P in Hilbert order. To find approximate nearest neighbors of a query point, both algorithms perform best-first search on this graph. The first algorithm uses standard one dimensional indexing structures to find starting points on the graph for this search, whereas the second algorithm using random starting points. Despite the quadratic preprocessing time, our algorithms have the potential to be useful in machine learning applications where the number of query points that need to be processed is large compared to the number of points in P. The linear dependence in d of the preprocessing and query time costs of our algorithms allows them to remain effective even when dealing with high-dimensional data.

1 The Problem

The K-Nearest Neighbors problem is the following: given a set P of N points in a $d-$dimensional space and a query point q, return the k points in P that are closest to q.

However, solving K-Nearest-Neighbors in high dimensions (say, more than 10) has proved computationally infeasible - most solutions are not much better than the naïve method. Thus, we consider the Approximate K-Nearest Neighbors problem: given a set P of N points in a $d-$dimensional space, a query point q, and parameters ϵ and δ between 0 and 1, return, with probability greater than $1-\delta$, k points of P such that the ith point is at most $(1+\epsilon)$ times farther from q than the true i^{th} nearest neighbor of q (Arya et al., 1994).

Approximate K-Nearest Neighbors is widely applicable, but we are motivated by its application to supervised machine learning. Machine learning applications are often characterized by a data set of relatively high dimensionality, so we are interested in solutions that scale well with d. In a typical supervised learning scenario, a training set is processed offline, and later the system must be able to quickly answer a stream of previously unknown queries. Our assumption is that the number of queries will be large compared to N, which is why we are more concerned with query time than preprocessing time. Many supervised machine learning techniques that could be alternatives to K-Nearest Neighbors have quadratic or cubic (in N) training time. To this end, our goal is to make query time as fast as possible, and accept almost any reasonable preprocessing cost (quadratic in N or better). Since the naïve algorithm has a query time complexity of $O(Nd)$, we demand a solution that provides query times sublinear in N and linear in d.

2 Related Work

Recent solutions to K-Nearest Neighbors that we have found tend to fall into two categories: ones that employ locality sensitive hashing (LSH) (Andoni and Indyk, 2008; Gionis et al., 1999) or ones that use sophisticated tree-like structures to do spatial partitioning (Liu et al., 2004; Arya and Mount, 1993; Arya et al., 1994; Beckmann et al., 1990; Berchtold et al., 1996). LSH defines a hash function on the query

space which has a collision probability that increases as the distance between two points decreases. In general, LSH approaches scale reasonably well with d, while the tree-based algorithms tend not to scale as well. Most notably, Arya et al. (1994) present an algorithm which, for fixed d, has a preprocessing cost of $O(N \log N)$ and a query time of $O(\log N)$, but is exponential in d. There are results which scale well with d and have fast query time. In particular, Kleinberg (1997) presents an algorithm with query time $O(N + d \log^3 N)$, and preprocessing cost quadratic in d, linear in N, and is $O(1/\log(\delta))$ in δ. Andoni and Indyk (2008) use LSH to achieve a query time of $O(N^{1/c^2+o(1)}d)$, and pre-processing cost of $O(N^{1+1/c^2+o(1)}d)$, where $c = (1+\epsilon)$.

3 Algorithm 1

Our two algorithms are similar. We will describe the first in its entirety, and then describe the changes we make to produce the second one.

3.1 Overview

`Algorithm 1` creates several auxiliary data structures to speed up query processing. The most important of these index structures is a graph, G, that holds all the N points in P. To create G, we first create a linked list containing all the points in P that is sorted in the Hilbert order. Then we add edges to the graph by linking points that are close together. The goal is to create a connected graph (starting with a linked list ensures connectedness) in which two points that are close in space will be very likely to be close on the graph. We also construct a small, constant number of one-dimensional search structures, specifically red-black trees, that order points based on their projections onto one-dimensional subspaces of our space. Given a query point q, the one dimensional search structures are used to obtain a small set of initial guess nodes in G. These are the nodes corresponding to points whose projections are close to the projection of q in the one-dimensional subspaces. Starting at these guess points, our algorithm searches G until $k+m$ nodes have been touched, for some constant m (assuming $k+m$ is greater than the number of guess points, otherwise we touch each guess point once). The nodes are sorted by their distance to q, and the first k are returned.

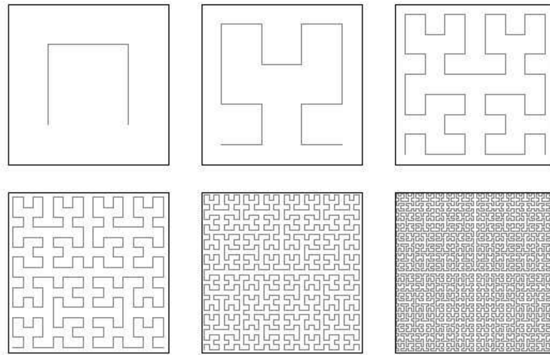

Figure 1: The first six curves in the sequence limiting to the Hilbert curve. Picture courtesy of Wikimedia.

3.2 Preprocessing and Auxiliary Data Structures

`Algorithm 1` requires several auxiliary data structures, as mentioned above. It uses a graph G on nodes corresponding to points in P, and several red-black trees. In order to create G, we first compute the Hilbert order of the points in P.

3.2.1 Snapping Points to an Integer Lattice and Sorting by Hilbert Order

The Hilbert curve is a space-filling curve defined as the limit of self-similar curves in a d-dimensional space. The first few curves in the 2-dimensional sequence are shown in Figure 1. One of the more celebrated properties of the Hilbert curve (Jagadish, 1990) is that it preserves locality well. That is, if two points are close in the d-dimensional space, their preimages on the unit interval will likely be close. Each curve in the sequence touches every point of a d-dimensional lattice with 2^n points on each side for some n. The Hilbert order of a set of points P on such a lattice is the order of the preimages of the points in the unit interval. We impose a Hilbert order on our set of points P by snapping them to such a lattice first. We compute the location in the lattice for a point by applying the following function to it:

$$f(\vec{x}) = \lceil a\vec{x} \rceil + \vec{b},$$

where

$$\frac{1}{a} = \min_{\vec{p},\vec{q} \in P} \left(\min_{i \leq d} |p_i - q_i| \right)$$

and

$$b_i = -\min_{\vec{p} \in P} \lceil p_i \rceil,$$

where x_i denotes the ith component of $\vec{x}$. That is, the smallest distance along any axis between any two

points becomes the lattice spacing. Such a potentially small lattice spacing could be undesirable because computing the Hilbert order might take too long. In practice, we have not found this to be a problem, but if it were, a coarser lattice could be used. Once the points are on a lattice, we compute the Hilbert order using an algorithm developed by Butz (1971), and explained to us by Jaffer (2008b). Our implementation is based on the implementation of `hilbert->int` in the SLIB library (Jaffer, 2008a).

3.2.2 Additional Graph Edges

The graph G begins as a linked list of the points of P in the Hilbert order as described above. For clarity, we will refer to nodes in G by the point they correspond to in P. Edges are strategically added as follows: each node p in G is linked to the b nodes closest to p in space, for some constant b. If p's b nearest neighbors are already adjacent to it (perhaps they were in the original linked list or they themselves have already been processed), these edges are not added again. This guarantees that each node of G will be adjacent to its b nearest neighbors. These nearest neighbors are computed using the naïve method, i.e., simply scanning all the points in P.

3.2.3 One-Dimensional Search Structures

In order to keep preprocessing costs low, we choose a subset P' of P consisting of $N^{2/3}$ points randomly selected from P. For some constant c, suppose the first c principal components of P' are $\{a_1, \ldots, a_c\}$. For each principal component a_i, we create a red-black tree T_i holding the elements of P ordered by where they fall along a_i.

3.3 Handling Queries

Given a query point q, we search each one-dimensional search structure T_i for the point p_i whose projection onto a_i is the closest to the projection of q onto a_i. These p_i are the c initial nodes for the search of G.

The search proceeds in a best-first manner by preferentially expanding nodes closer to q. If n is a node in G, let $d(n)$ denote the distance from n to q. Two priority queues are maintained during the search, *ToExpand* and *BestK*. We initialize *ToExpand* to contain the nodes p_i. *ToExpand* is a minheap with the priority of node n being $d(n)$. We initialize *BestK* to be empty. *BestK* is a maxheap, such that the highest priority node l maximizes $d(l)$. For $m + k$ steps, a node n is removed from *ToExpand.* If $d(n) > d(l)$ for the node l with the highest priority in *BestK,* then n is added onto *BestK* and l is removed (assuming *BestK* contains k items). Then all of the nodes adjacent to n are added to *ToExpand.* After $m+k$ steps, the k nodes in *BestK* are returned.

3.4 Cost Analysis

In order to compute the Hilbert order, we map each point in P to its distance from the origin along the appropriate Hilbert curve. This computation is $O(d + \log s)$, where s is a times the maximum coordinate of p, where a is the factor from our lattice snapping. Although we cannot control the maximum coordinate or a, we find that in practice, at least, we can compute the Hilbert order very quickly in hundreds of dimensions. We could theoretically control these variables by creating a coarser lattice, which might result in an approximation of the Hilbert order. We are convinced that this is not a problem. We assume that either s is reasonable or we force it to be so by adjusting the lattice, so this step should take time approximately linear in Nd.

We can complete the preprocessing phase of our algorithm in $O(N^2 d)$ time. Once points are in a linked list, for each point in P we add at most b additional edges. The new edges can be computed for a given point with a single scan of the points in P which will require $O(N)$ distance computations which each take time linear in d. Therefore we can construct the graph in $O(N^2 d)$ time. Computing the principal components of a subset of P of cardinality $N^{2/3}$ can be done in time quadratic in N since Principal Component Analysis can be performed in time cubic in the size of the dataset. The search trees can easily be constructed in $O(N \log N)$ time, so our preprocessing phase can be completed in $O(N^2 d)$ time. The space requirements of our auxiliary data structures are clearly linear in N.

To evaluate a query, we need to do a constant number of searches of red-black trees on N nodes which will have a cost logarithmic in N. We also have to project the query point into one dimension which adds a term proportional to d to our cost. In the best-first search of the graph we search a constant number of nodes and do a constant number of distance computations in d dimensions. Thus our query time is $O(d + \log N)$.

4 Algorithm 2

`Algorithm 2` is a simplification of `Algorithm 1`. Initial experiments suggested that the one-

dimensional search structures used in `Algorithm 1` were not that important to performance. Since the 1D search structures added a term proportional to $\log N$ to our query time, we designed a new algorithm that does not use them. In `Algorithm 1`, the red-black trees were used to get starting points for the best-first search. In `Algorithm 2`, we simply pick c random start points for this search. However, these start points will almost certainly be worse guesses than those produced by the red-black trees. Since we only expand a small constant number of nodes in our best-first search and since all of the edges in G connect points that are close in space, the search will expand many nodes that are far from the query point. Our solution is to add some longer edges to G. In the preprocessing phase of `Algorithm 2`, for each node in G, we add an edge from that node to one other random node (if the edge does not already exist). On average, these edges will be much longer than the nearest-neighbor edges.

4.1 Cost Analysis

The preprocessing phase of `Algorithm 2` is identical to the preprocessing phase of `Algorithm 1`, except

1. PCA is not performed.
2. Red-black trees are not constructed.
3. We add up to N random edges.

Adding the N random edges requires a single scan of the nodes in G, therefore our preprocessing time is still $O(N^2 d)$.

The term proportional to $\log N$ in the query time of `Algorithm 1` resulted from querying the red-black trees. `Algorithm 2` does not do this, so the query time for `Algorithm 2` is $O(d)$.

5 Experiments

Because we do not have proofs about the relationships between b, c, m, ϵ, and δ, we ran extensive empirical tests of our algorithms.

5.1 Datasets

We tested our algorithms on several large high dimensional data sets, both synthetic and real. Our synthetic data consists of mixtures of multivariate Gaussian distributions. Covariance matrices and means were generated randomly to produce these distributions. In particular, we considered unimodal, bimodal, pentamodal, and dodecamodal synthetic distributions. Gaussian distributions were chosen both because they often approximate distributions found in the wild, and because given a finite variance, the Gaussian distribution achieves the maximum entropy. We predicted that our algorithms would perform best on data with fewer modes, so the higher modal[1] distributions were selected to challenge our algorithms. In the case of the synthetic data, query points were drawn from the same distribution used to create the data. All of the synthetic data sets were 50-dimensional and contained 3000 points.

Real-world data was obtained from the UCI Machine Learning Repository (Asuncion and Newman, 2007). We used the ISOLET data set, which is audio data of spoken letters of the alphebet, and the Waveform Database Generator dataset. The ISOLET data set has 617 dimensions and more than 5000 points. The waveform data set has 22 dimensions and also more than 5000 points. In the case of the real data, the data sets were split into two parts, one for the initial set of points, and one from which to draw queries.

5.2 Parameters and Measurements

For each data set tested, the independent variables were:

- b: The number of nearest neighbors each point in P is connected to in G.
- c: The number of one-dimensional search structures created in the case of `Algorithm 1`, or the number of guess points in the case of `Algorithm 2`.
- m: The number of nodes (beyond k) in the graph that are expanded.
- k: The number of nearest neighbors requested.

The variables measured were:

- Percent Correct: The percent of the points returned which are actually among the k nearest neighbors.
- Excess Rank: The actual rank (that is, the j of "j^{th}-nearest neighbor") of the worst point returned, minus k.
- Maximum Epsilon: If the actual i^{th} nearest neighbor of a query point q has distance d_i from q, and the ith approximate nearest neighbor has distance d'_i from q, then Max Epsilon=

[1] With more extreme modality/modacitude, as it were.

$\max_i (d'_i/d_i - 1)$. Note that this is an upper bound on the ϵ from the definition of AkNN, if $\delta = 1$.[2]

Based on preliminary experiments, the parameters b, c, m, and k were allowed to vary in the ranges below.

- $b \in \{0, 1, \ldots, 10\}$
- $c \in \{1, 4, 16\}$
- $m \in \{0, 1, 10, 30, 60, 100, 150, 200, 250\}$ in `Algorithm 2`. For `Algorithm 1`, we ommitted 200 and 250 because m had less of an impact.
- $k \in \{100\}$

We ran preliminary experiments using a variety of values for k, but settled on $k = 100$. The relationships between parameters are easier to determine for larger k, since the algorithms are identifying more neighbors, and so setting $k = 100$ is more enlightening than looking at smaller values of k. For each combination of parameters, 50 queries were made, and the average value for each dependent variable was recorded. An implementation of the naïve exact kNN algorithm determined the accuracy of the approximate nearest neighbors. We ran experiments using the above parameters on all of our synthetic data sets.

We tested how each parameter affected our performance metrics (Percent Correct, Maximum Epsilon, Excess Rank) when the other parameters were small and fixed. We fixed two of $c = 4$, $m = 10$, $b = 4$, varying the third.

On the real-world data, we picked reasonable values for all parameters and did not vary them. The chosen parameter values were $c = 4$, $b = 4$, $m = 100$, $k = 100$.

In the results presented below, the graphs for Excess Rank exactly followed the graphs for Maximum Epsilon, so we omit them.

6 Results

Unsurprisingly, b is an important parameter for both algorithms. Figure 2 shows that for `Algorithm 1`, if c and m are small, b can easily pick up the slack. For $b > 4$, nearly all of the nearest neighbors are guessed correctly. As shown in Figure 3, for distributions with few modes, using a large enough b ensures

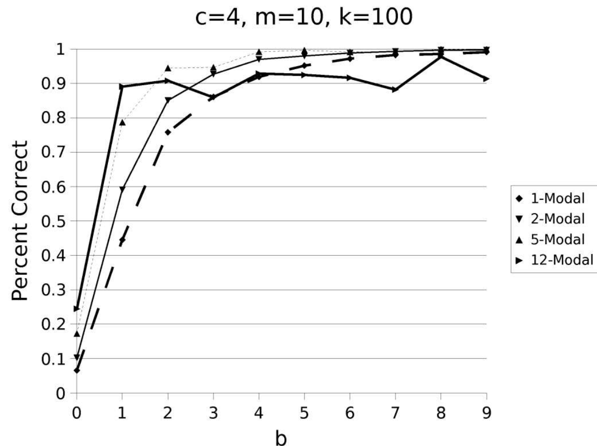

Figure 2: `Algorithm 1` on synthetic data sets with 3000 points in 50 dimensions. For $b > 4$, nearly 100% of the nearest neighbors are correctly identified.

that Maximum Epsilon is close to zero. Unfortunately, when we run `Algorithm 1` on distributions with more modes, Maximum Epsilon is not as close to zero.

As can be seen in Figure 4, `Algorithm 2` scales with b in the same way as `Algorithm 1` does. A high b guarantees a good maxEpsilon for the unimodal and bimodal cases, but the situation is worse for the 5 and 12−modal cases. This is because incorrect neighbors were sometimes drawn from a Gaussian in the mixture that was far away.

However, there is a difference in the Percent Correct achieved by our two algorithms as a function of b. As can be seen in Figure 5, while the general relationship between Percent Correct and b on a single distribution is the same for both algorithms, the distributions that are easier for `Algorithm 1` to handle are not the distributions that are easier for `Algorithm 2` to handle. While `Algorithm 1` had a lower Percent Correct on the 12-modal distribution, even for larger b, `Algorithm 2` appears to behave in the opposite way. For `Algorithm 2`, Percent Correct is highest for the 12-modal distribution for pretty much all b. In all cases, a choice of $b > 4$ still guaranteed a high Percent Correct.

Increasing m improves Percent Correct and Max Epsilon. However, over the ranges we tested, b has more of an impact than m on the performance of `Algorithm 1`. Figures 6 and 7 demonstrate this effect nicely. As we have come to expect for `Algorithm 1`, the 12-modal distribution produces the worst performance. `Algorithm 2` benefits even more than `Algorithm 1` from increased m. Figures 8 and 9

[2]Further note that this is not generally how the experimental ϵ is computed for AkNN.

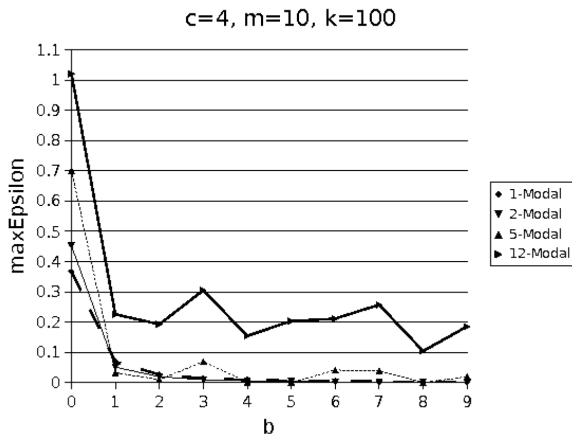

Figure 3: `Algorithm 1` on synthetic data sets with 3000 points in 50 dimensions. For $b > 4$ and for data sets with few modes, the points which are not correct are not far from the points which are correct. For data sets with many modes, this is less true.

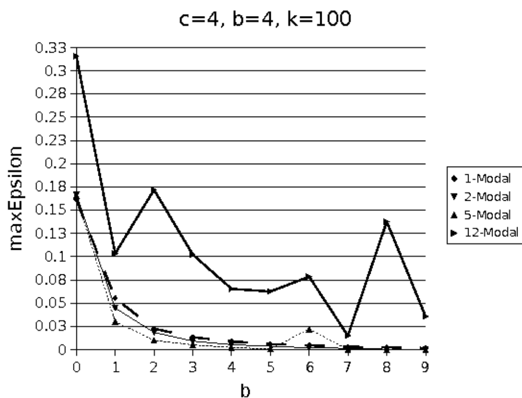

Figure 4: `Algorithm 2` on synthetic data sets with 3000 points in 50 dimensions. For $b > 4$ and for data sets with few modes, the points that are not correct are not far from the correct points. For data sets with more modes, this is less ture.

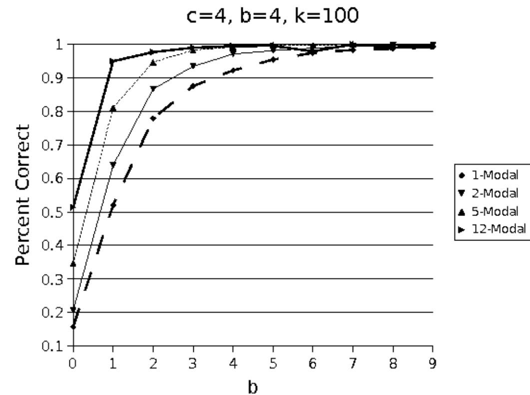

Figure 5: `Algorithm 2` on synthetic data sets with 3000 points in 50 dimensions. For $b > 4$, nearly 100% of the nearest neighbors are correctly identified. Surprisingly, `Algorithm 2` does better on more complicated distributions.

demonstrate that when we run `Algorithm 2` with increasing m on any of our synthetic data sets, Percent Correct rapidly approaches 1 and Max Epsilon rapidly approaches 0. The increased impact of m of `Algorithm 2` makes sense because `Algorithm 2` partly depends on a more extensive search to make up for its random starting points.

For both algorithms, over the ranges that we tested, c had less of an impact on performance than m or b. In particular, the Percent Correct for `Algorithm 2` was almost independent of c. It should be noted that while c represents the number of initial points for the search in both algorithms, these points are obtained in completely different ways. Thus, we do not gain much insight by comparing the effects on `Algorithm 1` and `Algorithm 2` of varying c. At some point, changing c would impact performance, but we are content to find a single reasonable value for c and focus on the other more important parameters.

Due to time constraints, we did not test `Algorithm 2` on our real world data sets. However, `Algorithm 1` performed admirably, especially considering the large number of dimensions (617) in the ISOLET data set. We have no reason to believe that `Algorithm 2` would not be as good or better than `Algorithm 1` on the real world data. Our experiments on synthetic data sets suggested that $m = 100$, $b = 4$ and $c = 4$ would be reasonable parameter settings that should work on any task. The results of running `Algorithm 1` with these parameters on all of our data sets are shown in Table 1. `Algorithm 1` returned more than 90% of the correct k nearest

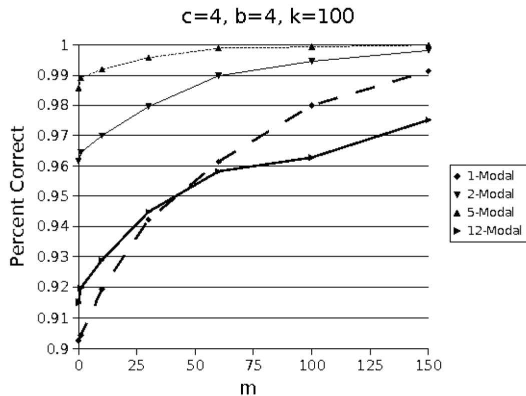

Figure 6: `Algorithm 1` on synthetic data sets with 3000 points in 50 dimensions. As we would hope, Percent Correct increases with m.

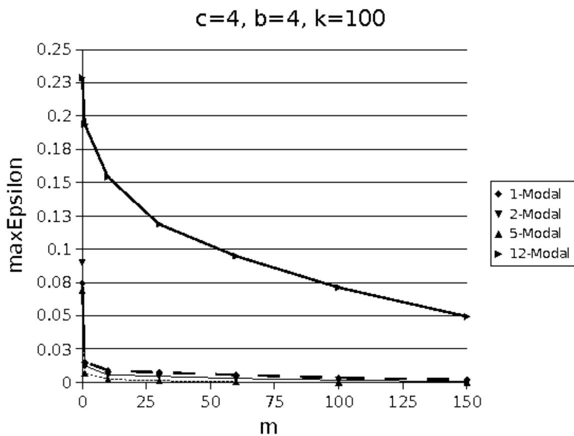

Figure 7: `Algorithm 1` on synthetic data sets with 3000 points in 50 dimensions. m seems relatively effective for data sets with few modes. For data sets with many modes, this is less true.

	PC	ME	ER
1-Modal	0.919	0.009	9.83
2-Modal	0.970	0.005	3.22
5-Modal	0.992	0.002	0.82
12-Modal	0.929	0.154	58.86
ISOLET	0.922	0.042	33.93
Wave	0.952	0.009	5.55

Table 1: Average Percent Correct (PC), Max Epsilon (ME), and Excess Rank (ER) over 100 queries on real world data (ISOLET and Wave) and over 50 queries on the synthetic data

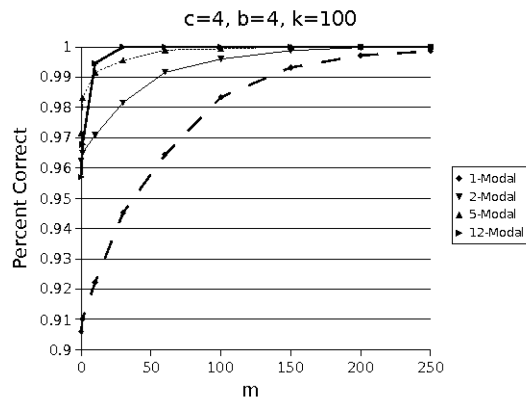

Figure 8: `Algorithm 2` on synthetic data sets with 3000 points in 50 dimensions. As we would hope, Percent Correct increases with m, and more so than in Figure 6.

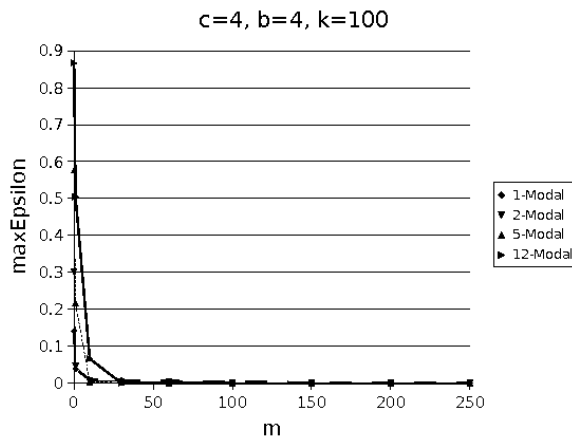

Figure 9: `Algorithm 2` on synthetic data sets with 3000 points in 50 dimensions. m seems relatively effective for data sets with any number of modes.

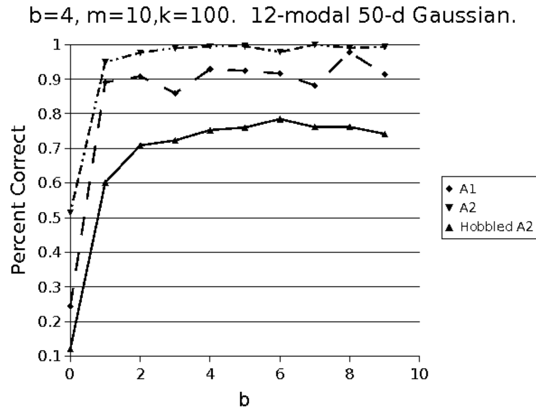

Figure 10: Both algorithms on the 12-modal synthetic data sets with 3000 points in 50 dimensions, along with a new algorithm `Hobbled Algorithm 2`. This algorithm is the same as `Algorithm 2`, except no random edges are added.

neighbors of the query points on each data set. As we have come to expect, `Algorithm 1` performed worse on the 12-modal data set. The ISOLET data set was also difficult, as presumably its intrinsic dimensionality was much larger that the dimensionality of the synthetic data.

6.1 Comparison of Algorithms

While `Algorithm 2` may sacrifice quality in its initial guesses when compared to `Algorithm 1`, it also has a more complete graph to search. Comparing the performance of both algorithms on our synthetic data sets, we found that `Algorithm 2` tended to outperform `Algorithm 1` on the 12-modal set, and that there was no discernable pattern on the other sets. One might wonder whether or not using the one dimensional search structures helped `Algorithm 1` at all, or whether the additional random edges helped `Algorithm 2`. Figure 10 demonstrates that the answer to both questions is yes. While `Algorithm 2` does slightly better than `Algorithm 1`, both do much better than `Hobbled Algorithm 2`, which is the same as `Algorithm 2`, except no random edges are added.

7 Conclusions and Future Work

The adjustments we made to `Algorithm 1` to obtain `Algorithm 2` suggest a third algorithm with $O(Nd)$ preprocessing time and $O(d)$ query time that might be interesting to try in future work. This algorithm, tentatively called `Algorithm 3`, would start with the same linked list as `Algorithms 1` and 2, add b random edges to each node, and process queries as in `Algorithm 2`. Dispensing with the nearest neighbor edges would give us the faster preprocessing time. It would be interesting to see how accurate `Algorithm 3` would be.

We have presented two algorithms which, though $O(N^2d)$ in preprocessing cost, handle queries in $O(d + logN)$ and $O(d)$ time, respectively, with experimentally good accuracy. As they are only linear in d, our algorithms scale well to the high-dimensional problems common in machine learning. Furthermore, our algorithms do not appear to be overly sensitive to parameter settings - choices of, say, $m = 100$, $b = 4$, and $c = 4$, seem to be sufficient to get good accuracy on all the data sets we tried. Our second, faster algorithm seems to do as well or better than our first algorithm, and its performance seems depend even less on the data distribution. Since `Algorithm 2` is faster and seems to be more robust, it should be preferred in general. Ultimately, our second algorithm is an attractive choice for solving Approximate K Nearest Neighbors in high dimensions.

References

Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122.

Arya and Mount. 1993. Approximate nearest neighbor queries in fixed dimensions. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*.

Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Wu. 1994. An optimal algorithm for approximate nearest neighbor searching. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 573–582, Philadelphia, PA, USA.

A. Asuncion and D.J. Newman. 2007. UCI machine learning repository.

Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331.

Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman,

Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A. Morgan Kaufmann Publishers.

A. R. Butz. 1971. Alternative algorithm for hilbert's space-filling curve. *IEEE Trans. Computers*, C-20:424–426.

Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Jaffer. 2008a. The SLIB Portable SCHEME Library, available at http://swissnet.ai.mit.edu/ jaffer/SLIB.html.

Aubrey Jaffer. 2008b. Personal Communication.

H. V. Jagadish. 1990. Linear clustering of objects with multiple attributes. *SIGMOD Rec.*, 19(2):332–342.

Jon M. Kleinberg. 1997. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 599–608.

T. Liu, A. Moore, A. Gray, and K. Yang. 2004. An investigation of practical approximate nearest neighbor algorithms.