Efficient similarity search and classification via rank aggregation

Ronald Fagin Ravi Kumar D. Sivakumar

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
{fagin, ravi, siva}@almaden.ibm.com

ABSTRACT

We propose a novel approach to performing efficient similarity search and classification in high dimensional data. In this framework, the database elements are vectors in a Euclidean space. Given a query vector in the same space, the goal is to find elements of the database that are similar to the query. In our approach, a small number of independent "voters" rank the database elements based on similarity to the query. These rankings are then combined by a highly efficient aggregation algorithm. Our methodology leads both to techniques for computing approximate nearest neighbors and to a conceptually rich alternative to nearest neighbors.

One instantiation of our methodology is as follows. Each voter projects all the vectors (database elements and the query) on a random line (different for each voter), and ranks the database elements based on the proximity of the projections to the projection of the query. The aggregation rule picks the database element that has the best median rank. This combination has several appealing features. On the theoretical side, we prove that with high probability, it produces a result that is a $(1+\epsilon)$-factor approximation to the Euclidean nearest neighbor. On the practical side, it turns out to be extremely efficient, often exploring no more than 5% of the data to obtain very high-quality results. This method is also database-friendly, in that it accesses data primarily in a pre-defined order without random accesses, and, unlike other methods for approximate nearest neighbors, requires almost no extra storage. Also, we extend our approach to deal with the k nearest neighbors.

We conduct two sets of experiments to evaluate the efficacy of our methods. Our experiments include two scenarios where nearest neighbors are typically employed—similarity search and classification problems. In both cases, we study the performance of our methods with respect to several evaluation criteria, and conclude that they are uniformly excellent, both in terms of quality of results and in terms of efficiency.

1. INTRODUCTION

The *nearest neighbor problem* is ubiquitous in many applied areas of computer science. Informally, the problem is: given a database D of n points in some metric space, and a query q in the same space, find the point (or the k points) in D closest to q. Some prominent applications of nearest neighbors include similarity search in information retrieval, pattern classification, data analysis, etc. The popularity of the nearest neighbor problem is due to the fact that it is often quite easy and natural to map the features of real-life objects into vectors in a metric space; questions like similarity and classification then become nearest neighbor problems. Since the mapping of objects into feature vectors is often a heuristic step, in many applications it suffices to find a point in the database that is *approximately* the nearest neighbor. These problems lead to fascinating computational questions; there is an extensive literature on efficiently computing nearest and approximately nearest neighbors. For some recent theoretical work, see [18, 16, 19]; for recent theoretical/applied work, see [13, 1, 12, 5, 4, 20].

In this paper, we propose a novel method for similarity search, classification problems, and other nearest-neighbor-search-based applications. Our method is built on two basic paradigms, *rank aggregation* [8] and *instance optimal algorithms* [11]. Our method satisfies the following two demanding, even conflicting, criteria: it is a robust generalization of nearest neighbors, and it admits algorithms that are extremely efficient and *database-friendly*.

The starting point for our work is the following simple idea. Suppose we are conducting nearest neighbor searches with a database D of n points in the d-dimensional space X^d (where X is the underlying set—reals, $\{0,1\}$, etc.), and are given a query $q \in X^d$. We may consider each coordinate of the d-dimensional space as a "voter," and the n database points as "candidates" in an election process. Voter j, for $1 \leq j \leq d$, ranks all the n candidates based on how close they are to the query in the j-th coordinate. This gives us d ranked lists of the candidates, and our goal is to synthesize from these a single ordering of the candidates; we are typically interested in the top few candidates in this aggregate ordering.

The *rank aggregation* problem is precisely the problem of how to aggregate the d ranked lists produced by the d coordinates. The history of this problem goes back at least two centuries, but its mathematical understanding took place in the last sixty years, and the underlying computational problems are still within the purview of active research [3, 14, 8]. The most important mathematical questions on rank aggregation are concerned with identifying *robust* mechanisms for aggregation; particularly noteworthy achievements in this field are the works of Young [21] and Young and Levenglick [22], who showed that a proposal of Kemeny [17] leads to an aggregation mechanism that possesses many desirable properties. For example, it satisfies the *Condorcet criterion*, which says that if there is a candidate c such that for every other candidate $\tilde{c}$, a majority of the voters prefers c to $\tilde{c}$, then c should be the winner of the election. Aggregation mechanisms that satisfy the Condorcet criterion and its natural extensions are considered to yield robust results that cannot be "spammed" by a few bad voters [8].

Kemeny's proposal is the following: given n candidates and d permutations $\tau_1, \tau_2, \ldots, \tau_d$ of these candidates, produce the permutation σ that minimizes $\sum_{i=1}^{d} K(\tau_i, \sigma)$, where $K(\tau, \sigma)$ denotes the *Kendall tau* distance, that is, the number of pairs $(c, \tilde{c})$ of candidates on which the rankings τ and σ disagree (one of them ranks c ahead of $\tilde{c}$, while the other ranks $\tilde{c}$ ahead of c). We will call this a *Kendall-optimal aggregation.* Unfortunately, computing a Kendall-optimal aggregation of even 4 lists is NP-complete [8], so one has to resort to approximation algorithms and heuristics.

Let us now explicate the connection between nearest neighbors and rank aggregation. As a simple but powerful motivating example, note that if the underlying space is $\{0, 1\}^d$ endowed with the Hamming metric, then each voter really produces a partial order; given a query q, the i-th voter partitions the database D into two sets $D_i^+ = \{x \in D \mid x_i = q_i\}$ and $D_i^- = \{x \in D \mid x_i \neq q_i\}$, ranking all of D_i^+ ahead of D_i^-. (The notions of Kendall tau distance and Kendall-optimal aggregation still remain meaningful, since they are based on comparing two candidates at a time.) It is not hard to see that in this case, the Kendall-optimal aggregation of the partial orders produced by the voters *precisely* sorts the points in the database in order of their (Hamming) distance to the query vector q. Considering also the fact that the nearest neighbor problems in several interesting metrics can be reduced to the case of the Hamming metric [19, 16, 6], we note that the rank aggregation viewpoint is, in general, at least as powerful as nearest neighbors. (We will provide even more compelling evidence shortly.)

On the other hand, we have taken a problem (the nearest neighbor problem) that can be solved by a straightforward algorithm in $O(nd)$ time and recast it as an NP-complete problem. Even some of the good approximation algorithms and heuristics for the aggregation problem (e.g., see [8]) take time at least $\Omega(nd + n^2)$. However, the confluence of two key factors rescues us from this dilemma. Firstly, we are interested only in the top few elements in the aggregate ordering, and not in the complete ordering of all database points. Secondly, in the context of finding top k winners in the aggregation, a heuristic based on median ranks turns out to admit an extremely efficient implementation. We turn to this next.

1.1 Median rank aggregation

While computing Kendall-optimal aggregations is unlikely to admit efficient algorithms, a polynomial-time computable ordering that is optimal in the *footrule* sense (details in Section 2) yields a factor-2 approximation to a Kendall-optimal ordering. Moreover, footrule-optimal aggregation has the following nice heuristic, which we will call *median rank aggregation*: sort all the points in the database based on the median of the ranks they receive from the d voters. This is a reasonable heuristic, since if the median ranks are all distinct, then this procedure actually produces a footrule-optimal aggregation [8]. Thus, we have reduced our problem (heuristically) to that of finding the database point with the best median rank (or the points with the top few median ranks).

Instead of viewing median rank aggregation only as a heuristic approximation to a Kendall-optimal aggregation, we consider it to be a natural rank aggregation approach in its own right. As we shall show in Section 2.1.1, median rank aggregation gives an optimal solution for a notion of distance similar to the footrule distance. Moreover, median rank aggregation has two desirable qualities, which we will now elaborate on.

Database friendliness and instance optimal algorithms. A strong argument for using median rank aggregation is its *database friendliness*. Specifically, we would like to propose a solution to the (approximate) nearest neighbor problem that has properties desirable in a database system. Ideally, one would like to avoid methods that involve complex data structures, large storage requirements, or that make a large number of random accesses. For example, these considerations immediately rule out the theoretically provably good methods [19, 16, 18]; even methods from the recent database literature [13, 1, 5] are encumbered with one or more of these problems. By contrast, median rank aggregation uses sorting as the only pre-processing step[1], needs virtually no additional storage, and performs virtually no random accesses. By avoiding random accesses, our method does not need indices that can locate the value of a coordinate of an element.

We now discuss an especially efficient approach to median rank aggregation. Let us pre-sort the n database points along each of the d coordinates. Given a query $q = (q_1, \ldots, q_d)$, we can easily locate the value q_i, for $1 \leq i \leq d$ in the i-th sorted list, and place two "cursors" in this location. Once the $2d$ cursors have been placed, two for each i, by moving one cursor "up" and one cursor "down,' we can now produce a stream that produces the ranked list of the i-th voter, one element at a time, and on demand[2]. That is, we think of the d voters as operating in the following online fashion: the first time the i-th voter is called, it will return the database element closest to q in coordinate i, the second time it will return the second closest element in coordinate i, and so on. Thus, effectively, we have an *online* version of the aggregation problem to solve.

The fact that we can easily produce online access to the d voters (with calls of the form "return the next most highly ranked element"), together with the fact that we would like to produce the candidate with the best median rank, suggests that it might be possible to identify this winner *without* even having to read the ranked lists in their entirety! Indeed, computing aggregations of score lists using an "optimal" number of sequential and random accesses to the lists—and hopefully without having to consult the lists completely—has attracted much work in recent database literature (e.g., [9, 11, 15, 2]—see also the references in [10]). We will design an algorithm in the spirit of the *NRA*, or "no random access," algorithm from [11]. The method of [11], applied to the online median-rank-winner problem, yields an exceedingly crisp algorithm that can be summarized in one sentence. *Access the ranked lists from the d voters, one element of every list at a time, until some candidate is seen in more than half the lists—this is the winner.* We will call this algorithm the MEDRANK algorithm. We shall show that MEDRANK is not just a good algorithm, but up to a constant multiple, it is the best possible algorithm on *every* instance, among the class of algorithms that access the ranked lists in sequential order. In fact, even if we allow both sequential and arbitrary random accesses, the algorithm takes time that is within a constant factor of the best possible on every instance. This notion is called *instance optimality* in [11]. We generalize the algorithm MEDRANK to find the top k objects in the natural way. For example, after the winner is found, we continue the algorithm by accessing the ranked lists until a second element is seen in more than half the lists—this is the number 2 element. We show that this generalized algorithm is also instance optimal.

[1] It is traditional not to charge nearest-neighbor algorithms for preprocessing steps, where data structures are set up. This is because in typical applications, the query-time efficiency is much more important than the cost of preparing the data structures.

[2] A somewhat similar use of cursors appears in [6], in the context of approximate nearest neighbors for the Hamming metric.

Approximate nearest neighbors. Median rank aggregation can be combined with another powerful idea that has often been considered in the nearest neighbor literature, since the pioneering work of Kleinberg [18]. The idea is that of projections along random lines in the d-dimensional space. Specifically, we show in Section 2, using a simple geometric lemma first noted in [18], that if we project the n database points (as well as the query point) into m dimensions, where $m = O(\epsilon^{-2} \log n)$, and then run algorithm MEDRANK on the projected data, then with high probability, the winner according to the MEDRANK algorithm is an ϵ-approximate nearest neighbor of the query point under the Euclidean metric. (We say that c is an ϵ-approximate nearest neighbor of q if, for every $\tilde{c} \in D$, we have $d(c, q) \leq (1 + \epsilon)d(\tilde{c}, q)$, where $d(\cdot, \cdot)$ denotes the Euclidean metric.)

1.2 Rank aggregation vs. nearest neighbors

We feel that rank aggregation is a new and robust paradigm for similarity search and classification. As we noted earlier, it is provably as powerful as nearest neighbors, and it has a very efficient implementation (with essentially no sequential accesses). We now point out another advantage of rank aggregation over nearest neighbors, in the context of databases. Consider a similarity search problem where the objects do not naturally fit in any natural metric space, such as a catalog of appliances, where the "features" may be categorical (eg., color), or may be numerical but where different coordinates have incompatible units (such as dollars versus inches). In these situations, it is extremely artificial and questionable to model the objects as points in a metric space where all coordinates have the same semantics. In these situations, the rank aggregation paradigm fits in naturally: when looking for objects similar to a query object, simply sort the database according to each feature (eg., by color preference, cost, etc.), and aggregate the rankings produced. Catalog searches are very common database operations, and our algorithm MEDRANK, suitably implemented, should result in an efficient and effective solution to this problem.

1.3 Organization

The rest of this paper is organized as follows. Section 2 presents the technical results concerning MEDRANK and related algorithms, and concludes with a formal description of the algorithms. Section 3 describes our experiments and presents their analysis. Our experiments included two of the primary applications of nearest neighbors—similarity search and classification. In both cases, we show that the aggregation approach yields excellent results, both qualitatively and in terms of efficiency. We make some concluding comments in Section 4.

2. FRAMEWORK AND ALGORITHMS

In the first part of this section, we describe the framework, including necessary preliminaries about rank aggregation and about instance optimal algorithms. There are two main technical results in this part: (1) a reduction from the ϵ-approximate Euclidean nearest neighbor problem to the problem of finding the candidate with the best median rank in an election where there are n candidates and $O(\epsilon^{-2} \log n)$ voters; and (2) a proof that algorithm MEDRANK, which makes only sequential accesses to the d ranked lists, makes at most a constant factor more accesses than *any* algorithm that uses sequential *and* random accesses to the lists, for *every* database and query. Thus, MEDRANK is instance optimal in the database model for computing the median winner, and also yields a provably approximate nearest neighbor.

2.1 Rank aggregation, nearest neighbors, and instance optimal algorithms

2.1.1 Preliminaries

Let σ and τ denote permutations on n objects; by $\sigma(i)$, we will mean the *rank* of object i under the order σ (lower values of the rank are "better"). Often we will say that i is ranked "ahead of" or "better than" or "above" j by σ if $\sigma(i) < \sigma(j)$. The *Kendall tau* distance between σ and τ, denoted by $K(\sigma, \tau)$, is defined to be the number of pairs (i, j) such that either $\sigma(i) > \sigma(j)$ but $\tau(i) < \tau(j)$ or $\sigma(i) < \sigma(j)$ but $\tau(i) > \tau(j)$. The *footrule distance* between σ and τ, denoted by $F(\sigma, \tau)$, is defined to be $\sum_i |\sigma(i) - \tau(i)|$.

Let $\tau_1, \tau_2, \ldots, \tau_m$ denote m permutations of n objects. A *Kendall-optimal aggregation* of $\tau_1, \ldots, \tau_m$ is any permutation σ such that $\sum_i K(\sigma, \tau_i)$ is minimized; similarly, a *footrule-optimal aggregation* of $\tau_1, \ldots, \tau_m$ is any permutation σ such that $\sum_i F(\sigma, \tau_i)$ is minimized. It is known [7] that $K(\sigma, \tau) \leq F(\sigma, \tau) \leq 2K(\sigma, \tau)$. It follows that if σ is a footrule-optimal aggregation of $\tau_1, \ldots, \tau_m$, then the total Kendall distance of σ from $\tau_1, \ldots, \tau_m$ (namely the quantity $\sum_i K(\sigma, \tau_i)$) is within a factor of two of the total Kendall distance of the Kendall-optimal aggregation from $\tau_1, \ldots, \tau_m$. Furthermore, although computing a Kendall-optimal aggregation is NP-hard, computing a footrule-optimal aggregation can be done in polynomial time via minimum-cost perfect matching [8]. Given permutations $\tau_1, \ldots, \tau_m$, we define for each object i the quantity

$$\text{medrank}(i) = \text{median}(\tau_1(i), \ldots, \tau_m(i)).$$

Thus, medrank assigns to each object its median rank. The following easy proposition, pointed out in [8], shows that in many cases, median rank aggregation gives a footrule-optimal aggregation.

PROPOSITION 1. *Let $\tau_1, \tau_2, \ldots, \tau_m$ denote m permutations of the same set of objects, If the median values medrank(i) are all distinct, then medrank is a permutation that is a footrule-optimal aggregation of $\tau_1, \ldots, \tau_m$.*

Even when the median ranks are not distinct, the next proposition says that median rank aggregation gives an optimal solution for a notion of distance similar to the footrule distance. Let f be a function that assigns a score to each object, and let τ be a permutation, both on the same set of objects. Define $M(f, \tau) = \sum_i |f(i) - \tau(i)|$, where the sum is taken over all objects i. Thus, M is similar to the footrule distance, except that f is a function that assigns scores, rather than a permutation.

PROPOSITION 2. *Let $\tau_1, \tau_2, \ldots, \tau_m$ denote m permutations of the same set of objects. Then medrank is a function f that minimizes $\sum_{j=1}^m M(f, \tau_j)$.*

PROOF. We wish to minimize the quantity $\sum_{j=1}^m M(f, \tau_j) = \sum_{j=1}^m \sum_i |f(i) - \tau_j(i)| = \sum_i \sum_{j=1}^m |f(i) - \tau_j(i)|$. It is clear that this last quantity is minimized by taking $f(i)$ to be that value x_i that minimizes $\sum_{j=1}^m |x_i - \tau_j(i)|$, for each object i. Thus, we can minimize for each object i separately, and obtain the overall minimum. Fix i, and let $y_j = \tau_j(i)$. Then we wish to find x_i that minimizes $\sum_{j=1}^m |x_i - y_j|$. But it is well known (and easy to prove) that $\sum_{j=1}^m |x_i - y_j|$ is minimized by taking $x_i = \text{median}(y_1, \ldots, y_m)$. Hence, $\sum_{j=1}^m M(f, \tau_j)$ is minimized by taking $f(i)$ to be the median rank of i. The proposition follows. $\square$

Let D be a database of n points in $\mathbf{R}^d$. For a vector $q \in \mathbf{R}^d$, a *Euclidean nearest neighbor of* q *in* D is any point $x \in D$ such that for all $y \in D$, we have $d(x, q) \leq d(y, q)$, where d denotes the usual Euclidean distance. For a vector $q \in \mathbf{R}^d$ and $\epsilon > 0$, an *ϵ-approximate Euclidean nearest neighbor of* q *in* D is any point

$x \in D$ such that for all $y \in D$, we have $d(x,q) \leq (1+\epsilon)d(y,q)$, where $d(\cdot,\cdot)$ denotes the usual Euclidean distance. Let $||\cdot||$ denote the Euclidean norm; thus, $d(x,y) = ||x-y||$.

2.1.2 An algorithm for near neighbors

The idea of projecting the data along randomly chosen lines in $\mathbf{R}^d$ was introduced in the context of nearest neighbor search by Kleinberg [18]. Specifically, consider a point $q \in \mathbf{R}^d$, and let $u, v \in \mathbf{R}^d$ be such that $d(v,q) > (1+\epsilon)d(u,q)$. Suppose we pick a random unit vector r in d dimensions; an efficient way to do this is to pick the d coordinates $r_1, \ldots, r_d$ as i.i.d. random variables distributed according to the standard normal distribution $N(0,1)$, and normalize the vector to have unit length. We then project u, v, and q along r. Let $\langle\cdot,\cdot\rangle$ denote the usual inner product. Then intuitively, we expect the projection $\langle u, r\rangle$ of u to be somewhat closer to the projection $\langle q, r\rangle$ of q than the projection $\langle v, r\rangle$ of v is. That is, we expect $\langle u-q, r\rangle$ to be smaller than $\langle v-q, r\rangle$. The following lemma implies a formal statement of this fact.

LEMMA 3 ([18]). *Assume $x, y \in \mathbf{R}^d$, and let $\epsilon > 0$ be such that $||y|| > (1+\epsilon)||x||$. If r is a random unit vector in $\mathbf{R}^d$ (chosen as described above), then* $\Pr[\langle y, r\rangle \leq \langle x, r\rangle] \leq 1/2 - \epsilon/3$.

By letting $x = u-q$ and $y = v-q$, it follows easily that $\langle u - q, r\rangle$ is smaller than $\langle v-q, r\rangle$ with probability at least $1/2 + \epsilon/3$.

Now let q be a query point, let $w \in D$ be the closest point to q, and let $B = \{x \in D \mid d(x,q) > (1+\epsilon)d(w,q)\}$. Consider a fixed $x \in B$. If we pick a random vector r and rank the points in D according to their distances from the projection of q along r, then w is ranked ahead of x with probability at least $1/2+\epsilon/3$. Suppose we pick several random vectors $r_1, \ldots, r_m$ and create m ranked lists of the points in D as follows: the j-th ranked list is obtained by sorting the points in D according to their projections along r_j. Then the expected number of lists in which w is ranked ahead of x is at least $m(1/2+\epsilon/3)$; indeed, by standard Chernoff bounds, if $m = \alpha\epsilon^{-2}\log n$ with α suitably chosen, then with probability at least $1 - 1/n^2$, we have that w is ranked ahead of x in more than $m(1/2+\epsilon/6)$ of the lists. Summing up the error probability over all $x \in B$, we see that this implies that with probability at least $1 - 1/n$, we have that w is ranked ahead of *every* $x \in B$ in more than $m(1/2+\epsilon/6)$ of the lists; in particular, the median rank of w in the m lists is better than the median rank of x in the m lists. Therefore, if we compute the point $z \in D$ that has the best median rank among the m lists, then (with probability at least $1-1/n$), we have that z cannot be an element of B, so it satisfies $d(z,q) \leq (1+\epsilon)d(w,q)$. We summarize this argument below.

THEOREM 4. *Let D be a collection of n points in $\mathbf{R}^d$. Let $r_1, \ldots, r_m$ be random unit vectors in $\mathbf{R}^d$, where $m = \alpha\epsilon^{-2}\log n$ with α suitably chosen. Let $q \in \mathbf{R}^d$ be an arbitrary point, and define, for each i with $1 \leq i \leq m$, the ranked list L_i of the n points in D by sorting them in increasing order of their distance to the projection of q along r_i. For each element x of D, let medrank(x) = median$(L_1(x), \ldots, L_m(x))$. Let z be a member of D such that medrank(z) is minimized. Then with probability at least $1-1/n$, we have $d(z,q) \leq (1+\epsilon)d(x,q)$ for all $x \in D$.*

In fact, the above argument shows more. Let q be a query, let $w \in D$ be the closest point to q, and let $w_2 \in D$ be the second closest point to q. Define the set $B_2 = \{x \in D \mid d(x,q) > (1+\epsilon)d(w_2,q)\}$ (notice that $B_2 \subseteq B$, where B is the set defined earlier to be $\{x \in D \mid d(x,q) > (1+\epsilon)d(w,q)\}$. By similar arguments, it follows that with high probability, the median rank of w_2 is better than that of any element in B_2; this implies that the element z_2 with the second best median rank must satisfy $d(z_2,q) \leq (1+\epsilon)d(w_2,q)$. Similarly, for any constant k, it can be shown that the elements $z_3, \ldots, z_k$ that achieve the third through the k-th best median ranks satisfy, respectively, $d(z_j,q) \leq (1+\epsilon)d(w_j,q)$, where w_j denotes the j-th closest element to the query q.

For the purposes of implementation, we cannot sort the n points of the database m times for each query q. Rather, as part of the preprocessing, we create m sorted lists of the n points in D. The i-th sorted list sorts the points based on the values of their projections along the i-th random vector r_i. The i-th sorted list is of the form $(c_1^i, v_1^i), (c_2^i, v_2^i), \ldots, (c_n^i, v_n^i)$, where (1) $v_t^i = \langle c_t^i, r_i\rangle$ for each t, (2) $v_1^i \leq v_2^i \leq \ldots v_n^i$, and (3) $c_1^i, \ldots, c_n^i$ is a permutation of $1, \ldots, n$. Given a query $q \in \mathbf{R}^d$, we first compute the projection of q along each of the m random vectors. For each i, we locate $\langle r_i, q\rangle$ in the (second coordinate of the) i-th sorted list, that is, find t such that $v_t^i \leq \langle r_i, q\rangle \leq v_{t+1}^i$, and initialize two cursors to v_t^i and v_{t+1}^i. One of points c_t^i and c_{t+1}^i is now the database point whose projection is closest to the projection of q. (This is the only step of the algorithm that will require random access.) By suitably moving one of the two cursors "up" or "down," we can implicitly create a list in which the database points are sorted in increasing order of the distance of their projections to q. This results in the following form of sequential access to the m lists: there is a routine that takes a query $q \in \mathbf{R}^d$ and initializes the $2m$ cursors, and there is a routine that returns the next element in the i-th list (in order order of proximity to the projection of q along r_i).

At the cost of more storage and pre-processing, we could also implement (full) random access to the sorted lists with indices. Then, given a point $x \in D$, this routine would return the rank of the point x in the i-th sorted list. Our algorithm MEDRANK does not need such random access.

2.1.3 Instance optimal aggregation

We have now reduced the problem of computing an ϵ-approximate nearest neighbor to the scenario of [11], which we now outline. There are m sorted lists, each of length n (there is one entry in each list for each of the n objects). Each entry of the i-th list is of the form (x, v_i), where v_i is the i-th "grade" of x. The i-th list is sorted in descending order[3] by the v_i value. In our case, v_i is simply the rank of object x in the i-th list (ties are broken arbitrarily). Further, there is an *aggregation function* [9, 11] that takes m scores and produces an "aggregate" value. The goal is to identify the k objects with the highest aggregate values.

There are two modes of access to data, namely sorted (or sequential) access and random access. Under sorted access, the aggregation algorithm obtains the grade of an object in one of the sorted lists by proceeding through the list sequentially from the top. Thus, if object x has the ℓ-th highest grade in the i-th list, then ℓ sorted accesses to the i-th list are required to see this rank under sorted access. The second mode of access is random access. Here, the aggregation algorithm requests the grade of object x in the i-th list, and obtains it in one random access.

In this scenario, our algorithm MEDRANK can be described as follows. The value v_i for object x is the rank of object x in the i-th list. The algorithm MEDRANK does sorted access to each list in parallel. The first object that it encounters in more than half the lists is remembered as the top object (ties are broken arbitrarily). The next object that it encounters in more than half the lists is remembered as the number 2 object, and so on until the top k objects have been

[3]In [11], the order is descending, which corresponds to the fact that bigger values are "better". For us, smaller values are better, since the values are ranks, and so we would (logically) sort in ascending order.

determined, at which time MEDRANK outputs the top k objects. Note that there are no random accesses (in our application of this algorithm for approximate nearest neighbors as outlined above, we incur random accesses during the initial setup of the cursors, but not subsequently). In fact, when the aggregation function is the median, it is easy to see that this algorithm is essentially the NRA ("No Random Access") algorithm of [11].

We shall show that in this scenario, algorithm MEDRANK is instance optimal [11], which intuitively corresponds to being optimal (up to a constant multiple) for every database. More formally, instance optimality is defined as follows. Let $\mathcal{A}$ be a class of algorithms, let $\mathcal{D}$ be a class of databases, and let *cost*(A, D) be the total number of accesses (sorted and random) incurred by running A on D.[4] An algorithm B is *instance optimal over* $\mathcal{A}$ *and* $\mathcal{D}$ if $B \in \mathcal{A}$ and if for every $A \in \mathcal{A}$ and every $D \in \mathcal{D}$ we have

$$\mathit{cost}(B, D) = O(\mathit{cost}(A, D)). \tag{1}$$

Equation (1) means that there are constants $c, c' > 0$ such that *cost*$(B, D) \leq c \cdot$ *cost*$(A, D) + c'$ for every choice of $A \in \mathcal{A}$ and $D \in \mathcal{D}$. The constant c is referred to as the *optimality ratio*.

In our case, $\mathcal{D}$ is the class of all databases consisting of m sorted lists, where the score of an object in each list is its rank in that list, and $\mathcal{A}$ is the class of all correct algorithms (that find the top k answers for the median rank) under our scenario (where only sorted and random accesses are allowed).

THEOREM 5. *Let $\mathcal{A}$ and $\mathcal{D}$ be as described above. Then algorithm* MEDRANK *is instance optimal over $\mathcal{A}$ and $\mathcal{D}$.*

PROOF. Assume that the algorithm MEDRANK, when run on $D \in \mathcal{D}$, halts and gives its output just after it has done ℓ sorted accesses to each list. Hence, the k-th lowest median rank is ℓ.

Let A be an arbitrary member of $\mathcal{A}$. Let us define a *vacancy* in the i-th list to be an integer j such that the object at level j in the i-th list was not accessed by algorithm A under either sorted or random access in the i-th list. Let U be the set of lists that have a vacancy at a level less than ℓ. We now show that the size of U is at most $\lfloor m/2 \rfloor$. Assume not. Define D' to be obtained from D by modifying each list in U as follows. Let x be a new object, not in the database D. For each list in U, the rank of x in that list is taken to be the level of the first vacancy in that list, and whatever object was in this position in that list in D is moved to the bottom of that list. Object x is placed at the bottom of each list not in U. Intuitively, x fills the first vacancy in each list in U. Since the rank of x is less than ℓ for more than half the lists, its median rank is strictly less than ℓ. Now algorithm A performs exactly the same on D and D', and so must have the same output. Therefore, algorithm A makes a mistake on D', since x is not in the top k list that A outputs, even though x has a median rank less than the median rank (ℓ) of some member of the top k list that A outputs. This is a contradiction, since by assumption A is a correct algorithm. So indeed, the size of U is at most $\lfloor m/2 \rfloor$.

Let Q be the number of accesses by A. From what we just showed, it follows that at least $\lceil m/2 \rceil$ lists have no vacancy at a level less than ℓ. This implies

$$Q \geq \lceil m/2 \rceil (\ell - 1) \geq (m/2)(\ell - 1).$$

Therefore, $m\ell \leq 2Q + m$. But $m\ell$ is the number of accesses performed by MEDRANK. Hence, MEDRANK is instance optimal, with optimality ratio at most 2. $\square$

[4]In [11], the cost of sorted and random accesses may be different. Taking the cost of all accesses to be the same, as we do here, affects the total cost by at most a constant multiple.

In the proof of Theorem 5, we saw that the algorithm MEDRANK has optimality ratio at most 2, with an additive constant of m. If we wish to get rid of the additive constant, we can use the fact that $Q > m/2$ to get an optimality ratio of 4, with no additive constant. It is interesting to note that the optimality ratios that are given in [11] are all linear or quadratic in m. Our algorithm is, as far as we know, the first nontrivial example in this context with an optimality ratio *independent* of m.

There are situations where algorithm MEDRANK probes the sorted lists more than halfway. However, it follows from results in [9] that when the lists are independently drawn at random, the expected probe depth of MEDRANK is $O(n^{1-2/(m+2)})$. When the sorted lists are positively correlated, we expect termination even earlier. In fact, when the rank lists are produced by computing proximity of the random projections of the database points to the corresponding projections of the query, it can be shown that the lists are significantly correlated. We omit the details.

We remark that MEDRANK is similar in an interesting way to Fagin's algorithm (FA) [9], in that FA halts only after seeing k objects in all m lists, whereas MEDRANK halts after seeing k objects in more than $m/2$ lists. In the case of FA, the k objects seen in all m lists need not be the top k objects, and so random accesses are still required. However, for MEDRANK, the k objects seen in more than $m/2$ lists are necessarily the top k objects, and so no random accesses are required.

2.2 Summary of algorithms

We now present formal descriptions of algorithm MEDRANK and of two related algorithms, OMEDRANK and L2TA. OMEDRANK is a heuristic improvement aimed at (further) improving its running time, and algorithm L2TA is an implementation of the "threshold algorithm" of [11], an instance optimal algorithm for computing Euclidean nearest neighbors in the model where data in each coordinate is accessed via sequential and random accesses. We will denote by L2NN the straightforward algorithm for finding nearest neighbors via a linear scan of all database elements.

The descriptions are in the standard "pseudo-code" style; where appropriate, we have decided in favor of clarity over fine, gory details and boundary conditions[5]. Also, we will describe the procedures to find the winner; the extensions to finding the top k elements are fairly straightforward.

We will assume that we have a database D of n points in $\mathbf{R}^m$, where $m = d$ (the original Euclidean space) or $m = O(\epsilon^{-2} \log n)$ (the space after projecting all data along m random lines). For $c \in D$ and $1 \leq i \leq m$, we will write c_i to denote the value of c in the i-th coordinate.

Algorithm MEDRANK is one among a family of aggregation algorithms, where we could strengthen the notion of median by considering quantiles other than the 50-th percentile. We introduce the parameter MINFREQ in MEDRANK to vary this value to the other quantiles. Even though the algorithms with other values of MINFREQ do not ostensibly have any connection to nearest neighbors, we expect them to be excellent aggregation algorithms as well. The MINFREQ parameter is a strict lower bound on the number of lists an element has to appear in before it is declared the winner. Taking the median rank corresponds to setting MINFREQ = 0.5. The idea is that by increasing MINFREQ, we can expect to improve the quality at the cost of more probes into the database, thereby permitting a quality–time tradeoff.

[5]For example, when we decrement or increment pointers, we do not make explicit what to do when they go out of range; similarly, when we create auxiliary index structures or sorted tables, we will not explicitly say how they are stored (B-tree or flat array, etc.).

Algorithm MEDRANK

Pre-processing

Create m lists $L_1, \ldots, L_m$, where L_i consists of the pairs (c, c_i) for all $c \in D$.

For $1 \leq i \leq m$, sort L_i in ascending order of the second component. Now each L_i has the form $(c_{i,1}, v_{i,1}), \cdots, (c_{i,n}, v_{i,n})$, where the $c_{i,t}$'s are the n distinct objects in the database, and $v_{i,1} \leq v_{i,2} \leq \ldots \leq v_{i,n}$.

Query-processing

Given $q \in \mathbf{R}^m$, for each i, initialize two pointers h_i and l_i into L_i so that $v_{i,h_i} \leq q_i \leq v_{i,l_i}$.

S will be a set of "seen elements" $c \in D$ and their frequencies f_c; initialize S to $\emptyset$.

```
while $S$ has no element $c$ s.t. $f_c >$ MINFREQ $* m$ do:
    for $1 \leq i \leq m$ do:
        if $|v_{i,h_i} - q_i| < |v_{i,l_i} - q_i|$ then
            set $c = c_{i,h_i}$ and decrement $h_i$
        else
            set $c = c_{i,l_i}$ and increment $l_i$
        if $c \notin S$, then
            add $c$ to $S$ and set $f_c = 1$
        else
            increment $f_c$
    end-for
end-while
```

Output the element $c \in S$ with the largest f_c.

The second algorithm we describe, OMEDRANK, is based on the following observation about MEDRANK (the notations are as in the description of the algorithm MEDRANK). Instead of comparing the values v_{i,h_i} and v_{i,l_i} and choosing the one closer to q_i, we will consider *both* elements c_{i,h_i} and c_{i,l_i}. Since we do not perform any random accesses (eg., "find the rank of c_{i,h_i} in some other list L_j"), this will increase the number of elements we consider for membership in S, but we save on many comparisons.

Algorithm OMEDRANK

Pre-processing

Identical to MEDRANK.

Query-processing

Given $q \in \mathbf{R}^m$, for each i, initialize two pointers h_i and l_i into L_i so that $v_{i,h_i} \leq q_i \leq v_{i,l_i}$.

S will be a set of "seen elements" $c \in D$ and their frequencies f_c; initialize S to $\emptyset$.

```
while $S$ has no element $c$ s.t. $f_c >$ MINFREQ $* m$ do:
    for $1 \leq i \leq m$ do:
        for $c \in \{c_{i,h_i}, c_{i,l_i}\}$ do:
            if $c \notin S$, then
                add $c$ to $S$ and set $f_c = 1$
            else
                increment $f_c$
        end-for
        decrement $h_i$ and increment $l_i$
    end-for
end-while
```

Output the element $c \in S$ with the largest f_c.

Finally, we describe an instance optimal algorithm for computing Euclidean nearest neighbors; this algorithm is an application of the "threshold algorithm," or TA, of [11] to the problem of computing Euclidean (or L_2) nearest neighbors. This algorithm, which we will call L2TA, can be used in place of the naive nearest neighbors algorithm, and is often much faster.

Algorithm L2TA

Pre-processing

Create m lists $L_1, \ldots, L_m$, where L_i consists of the pairs (c, c_i) for all $c \in D$.

For $1 \leq i \leq m$, sort L_i in ascending order of the second component. Now each L_i has the form $(c_{i,1}, v_{i,1}), \ldots, (c_{i,n}, v_{i,n})$, where the $c_{i,t}$'s are the n distinct objects in the database, and $v_{i,1} \leq v_{i,2} \leq \cdots \leq v_{i,n}$.

Create the index P such that $P(i, c)$ equals that value of j where $c_{i,j} = c$, for each $c \in D$ and $1 \leq i \leq m$. That is, $P(i, c)$ is the position (or rank) of c in the sorted list L_i.

Query-processing

Given $q \in \mathbf{R}^m$, for each i, initialize two pointers h_i and l_i into L_i so that $v_{i,h_i} \leq q_i \leq v_{i,l_i}$.

S will be a set of "seen elements" $c \in D$ and their distances d_c to q; initialize S to $\emptyset$.

T will be a "threshold value" that tracks the minimum distance that any unseen element $\tilde{c} \notin S$ can achieve to q; initialize T to 0.

```
while $S$ has no element $c$ s.t. $d_c \leq T$ do:
    for $1 \leq i \leq m$ do:
        let $a_i = |v_{i,h_i} - q_i|$ and $b_i = |v_{i,l_i} - q_i|$
        if $a_i < b_i$ then
            set $c = c_{i,h_i}$ and decrement $h_i$
        else
            set $c = c_{i,l_i}$ and increment $l_i$
        if $c \notin S$, then
            $s = 0$
            for $1 \leq j \leq m$ do:
                $p = P(j, c)$; $s = s + v_{j,p}^2$
            end-for
            add $c$ to $S$ and set $d_c = \sqrt{s}$
        end-if
    end-for
    $T = (\sum_{i=1}^{m} \min(a_i, b_i)^2)^{1/2}$
end-while
```

Output the element $c \in S$ with the smallest d_c.

3. EXPERIMENTS

3.1 Data collection

Our experimental setup consists of two data sets, which we call STOCK and HW respectively. In the following, we describe these data sets in detail. Note that by using a feature vector in a suitable high-dimensional space for each object, one can intepret similarity/classification problems as nearest-neighbor problems.

The first data collection STOCK was derived from the historical stock prices of several U.S. companies. The data was first collected from Yahoo!'s business page and consisted of the entire recorded history of stock prices of 7999 companies, excluding mutual funds. Each company's data was then split into periods representing 100 consecutive trading days (if there was any remainder the least recent data was discarded). We assumed that $1 was invested in the stock in the beginning of the trading period, and tracked the progress of this one dollar through the 100-day trading period. This created a feature vector in 100 dimensions. This process, repeated for each company, resulted in a data set of 145,619 vectors, where each feature vector resides in a 100-dimensional space. This way of partitioning the data was done for two reasons: first, to increase the size of the data set while maintaining nontrivial dimensionality; and second, to be able to compare different stocks over reasonable "windows" of time.

The second data collection was derived from the publicly available MNIST database of handwritten digits (at the web site
`http://yann.lecun.com/exdb/mnist`).
The original data consisted of a training set of 60,000 labeled examples and a test set of 10,000 examples. Each greyscale image was of size 28x28 and the labels were from 0 to 9. The feature vector of each image was just the 784 pixel values. Thus, each vector resides in a 784-dimensional space. Since we are interested only in nearest-neighbor-based classification (and not training), we collapsed these two sets into a single data set consisting of 70,000 vectors.

These two data sets were chosen to be contrasting in more than one aspect in order to add sufficient diversity to our experiments. While STOCK is a large data set with moderate dimension, HW is a relatively smaller data set, but with larger dimension. Another important distinction is that the data set HW is implicitly clustered, since it arose from 10 underlying classes. On the other hand, no natural clustering semantics can be easily associated with STOCK. Finally, the problems considered for each data set are different—similarity searching for the STOCK data, and classification performance for the HW data. Note that character recognition, as in the HW dataset, is an arguably important application of *approximate* nearest neighbors: it is more important here to classify the characters correctly than it is to find the exact nearest neighbor.

We decided to conduct all the experiments by storing the entire data in the main memory, and so hardware limitations prevented us from working with larger data. Notice, however, that by forcing ourselves to hold all the data in main memory, we are only helping the L2NN and L2TA algorithms, which we will compare MEDRANK and OMEDRANK against. If most of the data were to reside on secondary storage, these algorithms would be far more expensive, as they are prone to accessing a large fraction of the database, which might result in increased disk access.

3.2 Setup

To study the performance of the algorithms in reduced dimensions, we performed random projection on the data. For STOCK, we projected the data into dimensions DIM = 10, 20, 30, 40, 50. For HW, we projected the data into dimensions DIM = 20, 40, 60, 80, 120, 160, 200. For HW, the label of each vector was collected as well to determine the classification error. The parameter MINFREQ was chosen to be 0.5, 0.6, 0.7, 0.8, 0.9; this parameter has influence on the probe depth of MEDRANK and OMEDRANK.

We implemented the algorithms L2NN, L2TA, MEDRANK, and OMEDRANK in C++. Our experiments were run on a 1GHz Pentium machine with about 0.5G RAM. Note that our choice of the data set ensures that it will fit entirely in main memory.

For the original dimension and each of the reduced dimensions, and for each value of MINFREQ, the algorithms were run on 1000 queries on both STOCK and HW. The queries were selected at random from the same data set. Various parameters, described below, were averaged over these 1000 queries. (When considering query q, we implicitly were searching the database $D \setminus \{q\}$.)

3.3 Parameters studied

(1) *Time*. We study the basic running time of the algorithm to compute the top 10 results. The running time includes query-specific preprocessing. Since L2NN on the full dimensional data can be considered a reasonable approximation to the "absolute truth," we compare the running time of each algorithm relative to the running time of L2NN on the full dimensional data. (Even though L2TA on the full dimensional data is an "exact" algorithm, it is considerably slower than L2NN; hence, we compare the running times of all algorithms to L2NN rather than to L2TA.)

(2) *Quality.* We use two different notions of quality for STOCK and HW. For STOCK, it is the following. Let q be the query, let p be the (top) point in the data set returned by the algorithm (possibly using projected data) for the query q, and let p^* be a point in the data set returned by L2NN on the full dimensional data for the same query q. Intuitively, p^* is the "right answer" (an actual nearest neighbor). The quality is defined to be the ratio $d(p,q)/d(p^*,q)$.

In the case of HW, the quality is defined to be the following. Recall that we have collected the labels for HW data. Let ϵ be the classification error of our algorithms (possibly using projected data) for a set of queries and let ϵ^* be the classification error of L2NN on the full dimension data for the same set of queries; here, classification error is the fraction of queries on which the label returned by the algorithm differs from the true label of the query.[6] The quality is then defined to be the ratio ϵ/ϵ^*. The main reason for this, rather than presenting the absolute classification error, is that the classification error is not only a function of the nearest neighbor or aggregation algorithm, but also a function of the underlying feature set. (We have not attempted to optimize the quality of the underlying features; that is outside the scope of our work. We shall, therefore, restrict ourselves to comparing against the best that a brute-force nearest neighbor algorithm can achieve.)

Thus both these quantities are defined relative to the performance of L2NN on the full dimension data.

(3) *Probe depth and fraction accessed.* Recall that algorithms L2TA, MEDRANK, and OMEDRANK do not access the complete database in general. For MEDRANK and OMEDRANK which access the database in a (database-friendly) sequential manner, we record the number of such sequential accesses. In fact, we record the number of such accesses to output each of the top 10 results.

We anticipate the probe depth to be correlated with the expected rank of the closest point in the database in each of the m lists. (We talk about the expectation, since the m lists were produced probabilistically.) We computed the distribution of the quantity rank(w), where w is the "winner" for a query q (recall that we consider q as a query for the database $D \setminus \{q\}$). The distribution was computed by averaging the quantities over 1000 random queries. Figure 1 presents the distribution; the expectation of rank(w) (for the STOCK data) is roughly 0.13, which means that we expect MEDRANK and OMEDRANK to probe only 13% of the data on the average!

The algorithm L2TA, in addition to sequential accesses, also makes random accesses. We record this information as well.

3.4 Results

To avoid inundating the reader with too many numbers, we present only a subset of the basic results of the experiments on STOCK and HW in Tables 1 and 2 respectively.

(1) *Time.* As can be seen from the tables, even on full dimensional data, the running times of MEDRANK and OMEDRANK are substantially smaller than that of L2NN (roughly only 35–45% of

[6]The dataset contains the true class labels.

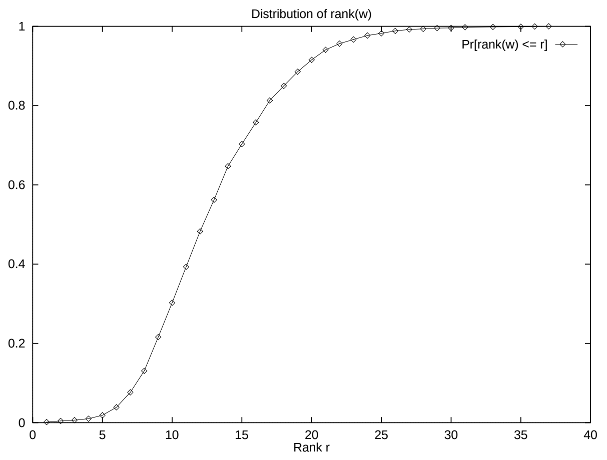


Figure 1: Distribution of rank(w).

the time taken by L2NN). On projected data, MEDRANK and OMEDRANK are faster by two orders of magnitude. These algorithms remain much faster than L2NN even at very high values of MINFREQ.

We remark that this difference would be even more pronounced were the data accessed from disk. Moreover, if we had counted the running time as the time to compute the top result (instead of the top 10 as we do now), MEDRANK and OMEDRANK would have performed even more dramatically.

Algorithm L2TA offers a significant speed-up at low dimensions for the STOCK data, but is poorer at high dimensions, and consistently worse than L2NN for the HW data. This can be attributed to the bookkeeping efforts in the algorithm.

	L2NN	L2TA		MEDRANK		OMEDRANK	
DIM	Time	Time	Qual.	Time	Qual.	Time	Qual.
				MINFREQ = 0.5			
10	0.195	0.065	1.399	0.002	1.794	0.004	1.790
20	0.289	0.139	1.270	0.005	1.518	0.006	1.514
30	0.376	0.232	1.231	0.008	1.430	0.009	1.426
40	0.466	0.344	1.201	0.013	1.338	0.013	1.332
50	0.555	0.440	1.186	0.017	1.333	0.015	1.330
100	1.000	11.00	1.000	0.459	1.360	0.352	1.434
				MINFREQ = 0.7			
10	0.195	0.065	1.399	0.003	1.654	0.004	1.663
20	0.289	0.139	1.270	0.007	1.414	0.009	1.412
30	0.376	0.232	1.231	0.012	1.344	0.013	1.345
40	0.466	0.344	1.201	0.020	1.273	0.018	1.274
50	0.555	0.440	1.186	0.026	1.264	0.023	1.259
100	1.000	10.99	1.000	0.817	1.253	0.645	1.286

Table 1: Basic performance measures for the algorithms on STOCK data at MINFREQ = 0.5, 0.7. Time denotes the time relative to L2NN in full dimensions and qual. denotes the distance ratio relative to the one obtained by L2NN in full dimensions.

(2) *Quality.* Again, the tables demonstrate that the quality of MEDRANK and OMEDRANK is high. For STOCK data, the factor of approximation is around 2, meaning that the closest point found by these algorithms is at most a factor of 2 away from the optimum. Note that L2TA will actually find the nearest neighbor and therefore match the quality of L2NN for that dimension. A more important

	L2NN	L2TA		MEDRANK		OMEDRANK	
dim.	Time	Time	Qual.	Time	Qual.	Time	Qual.
					MINFREQ = 0.5		
20	0.042	0.236	11.38	0.004	23.75	0.004	23.25
40	0.063	0.616	6.042	0.010	12.50	0.011	14.17
60	0.087	1.030	3.875	0.019	10.47	0.018	10.00
80	0.110	1.458	3.625	0.029	7.917	0.026	7.167
100	0.134	1.876	3.542	0.040	7.083	0.033	6.625
120	0.156	2.319	3.333	0.052	6.667	0.042	5.208
160	0.203	2.400	2.830	0.078	4.583	0.063	4.583
200	0.250	-	-	0.098	4.583	0.083	4.167
					MINFREQ = 0.9		
20	0.042	0.236	11.38	0.011	14.58	0.012	13.25
40	0.063	0.616	6.042	0.029	7.500	0.029	7.708
60	0.087	1.030	3.875	0.051	5.833	0.047	5.125
80	0.110	1.458	3.625	0.078	5.000	0.067	5.000
100	0.134	1.886	3.542	0.106	7.083	0.086	4.250
120	0.156	2.319	3.333	0.137	5.833	0.108	3.583
160	0.203	2.400	2.830	0.197	3.750	0.160	3.750
200	0.203	-	-	0.253	3.750	0.208	3.750

Table 2: Basic performance of various algorithms on HW data at MINFREQ = 0.5, 0.9. Time denotes the time relative to L2NN in full dimensions and Qual. denotes the classification error relative to the one incurred by L2NN on full dimensions.

point to notice is that a factor-2 approximation to the nearest neighbor is found at an amazingly low (often less than 1%) running time. The improvements are somewhat less dramatic for the HW data: at about 6% of the running time, we are able to achieve an error that is roughly 5 times more.

3.4.1 Other statistics

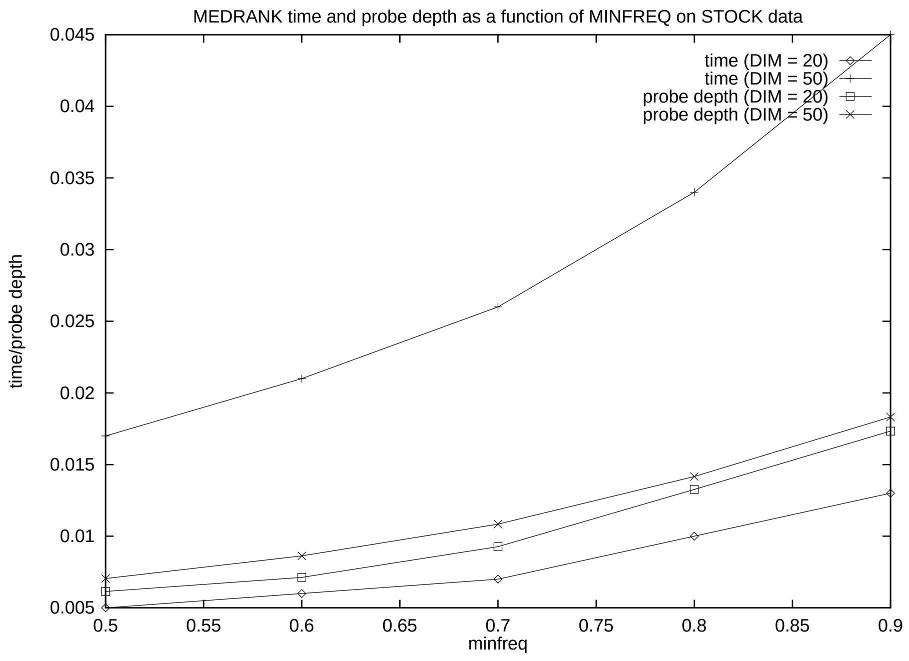
We present some other statistics about algorithms in Figures 2–5. These include: the effect of the MINFREQ parameter on the time, the depth to which the algorithms probe the database and its correlation to the running time, the quality of the solutions produced by the MEDRANK algorithm as a function of the MINFREQ parameter and the "top k" parameter, and the probe depth and access statistics for the L2TA algorithm. The captions for the figures list the main observations.

3.4.2 Inferences

(1) Both MEDRANK and OMEDRANK are extremely fast and scan only an extremely small portion of the database even when MINFREQ is increased to 0.9 (see Figure 2). Thus, these algorithms are very database friendly and represent an extremely efficient and effective alternative to L2NN.

(2) Projecting the data into lower dimensions is always an advantageous step, if one only cares about approximate nearest neighbors. While preserving correlations, random projection reduces the effects of noise. On projected data (our case of greatest interest), the quality of these algorithms almost matches that of L2NN on the same data, while the running times are significantly better. Projection also significantly reduces—by at least an order of magnitude—the depth of probes of these algorithms (see Figure 5). We conclude that while projection is a good idea if one is satisfied with an approximate nearest neighbor, MEDRANK and OMEDRANK are far better alternatives to L2NN (or even L2TA) on the projected data.

(3) Comparing MEDRANK and OMEDRANK, we note that in several cases, OMEDRANK offers up to 20% speed-up over MEDRANK, while preserving the quality of results (see Tables 1 and 2).



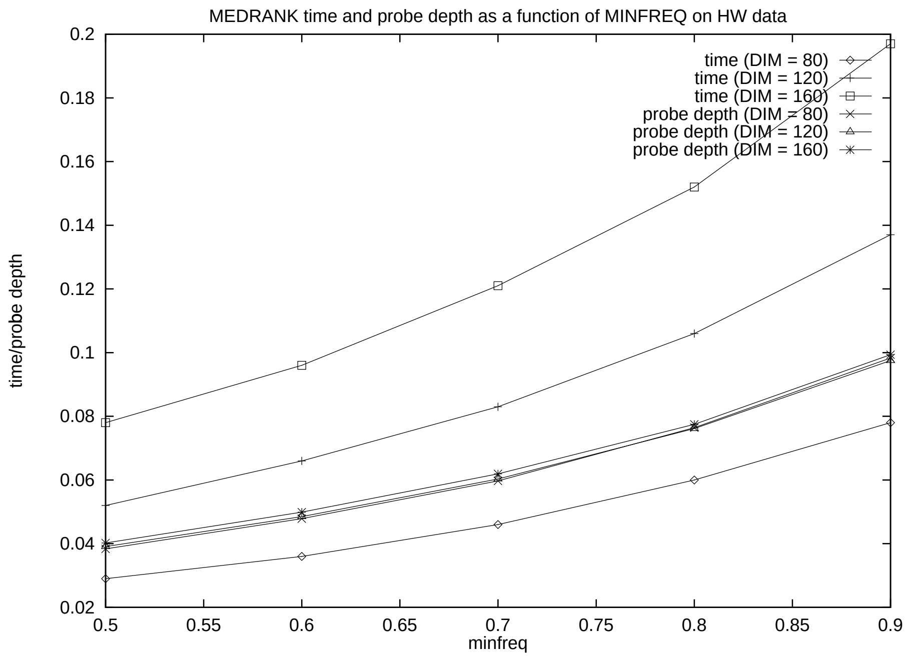


Figure 2: MEDRANK **time and probe depth as a function of** MINFREQ **on** STOCK **and** HW **data. Notes: (1) In both cases, dimension has almost no effect on the probe depth. (2) Even at** MINFREQ $= 0.9$**, time taken is very small.**

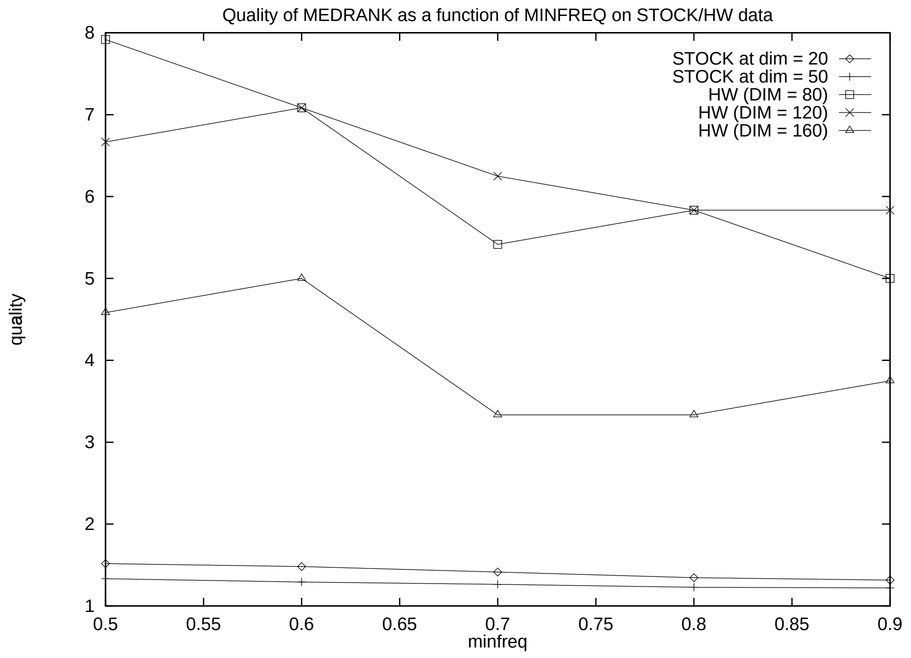


Figure 3: Quality of MEDRANK as a function of MINFREQ on STOCK/HW data. Notes: (1) HW data shows much more improvements as a function of dimensionality than STOCK data. (2) MINFREQ seems not to affect results on STOCK data much, but on the HW data, a value of roughly 0.7 seems to be the best.

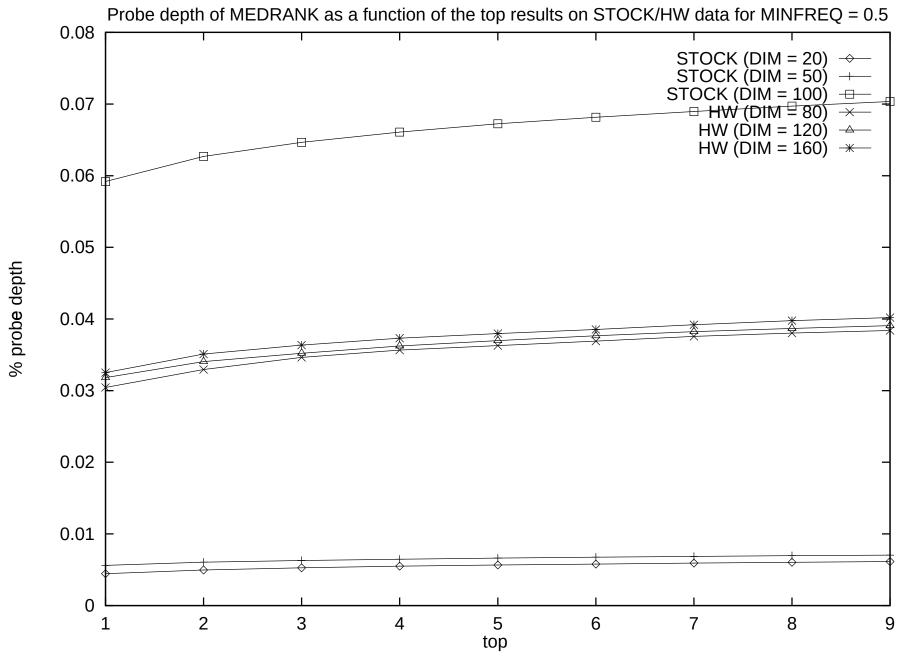
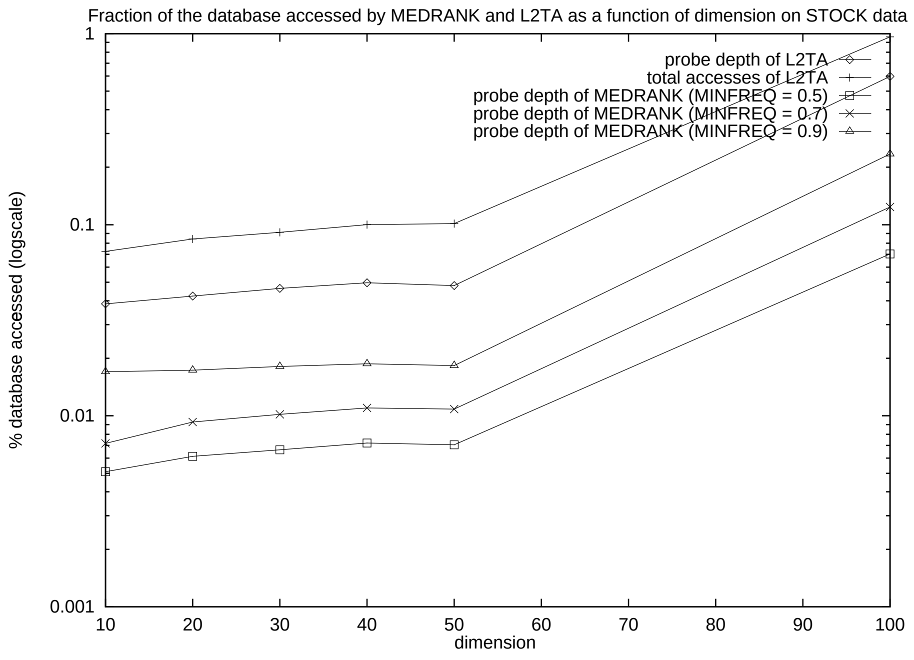


Figure 4: Probe depth of MEDRANK as a function of the top results on STOCK/HW data for MINFREQ=0.5. Notes: (1) Dimensionality reduction causes significant improvement in the probe depth for STOCK data (compare 100 dimensions vs. lower dimensions). Note that we did not conduct the HW data experiments on the full 784 dimensions. (2) Whether we are computing top 1 or top 10 seems not to affect probe depth by much in both cases.



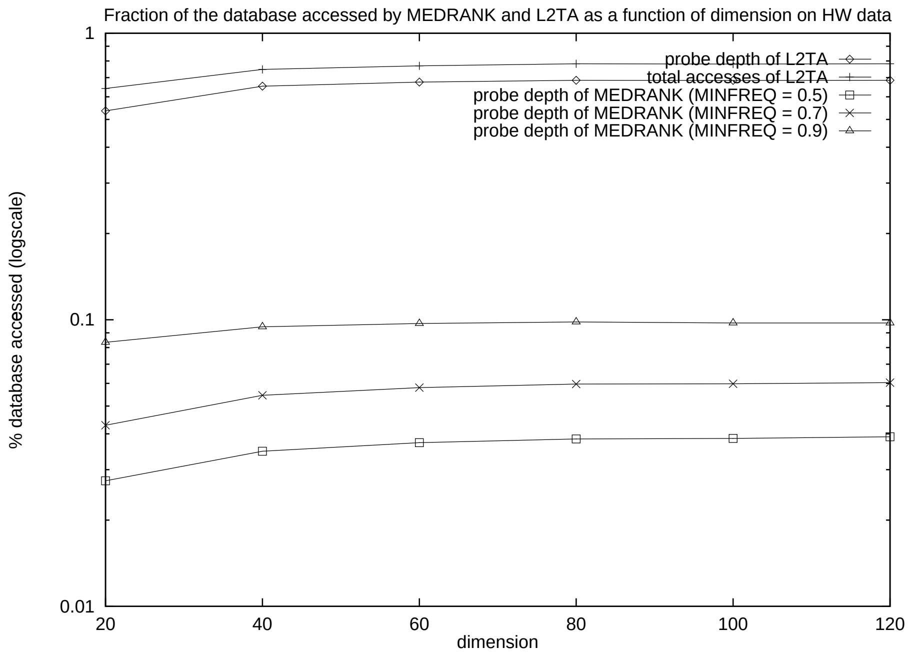


Figure 5: Fraction of the database accessed by L2TA **and** MEDRANK **as a function of dimension on** STOCK **and** HW **data. Note: (1)** MEDRANK **and** OMEDRANK **access an order of magnitude fewer elements of the database than** L2TA**.**

(4) The parameter MINFREQ has a varying role in terms of its significance to MEDRANK and OMEDRANK. For STOCK, we note that this parameter plays no significant role; therefore, it suffices to keep it low (at 0.5), which yields excellent running times. For HW, it contributes to lowering the error (see Figure 3). However, as one would suspect, it affects the probe depth (and therefore the running time) of these algorithms (see Figure 2). Yet, the probe depth still remains one or two orders of magnitude smaller than the size of the database, pointing to the robustness of these algorithms.

(5) We examine the question of how far MEDRANK has to go to uncover each of the top 10 results it produces. Figure 4 shows this as a function of the top results. As can be seen, there is not much difference between obtaining the top 1 and the top 10 results.

(6) We conclude that L2TA for the nearest neighbor problem offers nontrivial but not a dramatic improvement in speed at lower dimensions, and tends to become poor as the dimension increases (see Tables 1 and 2). Further confirming this is Figure 5 which shows the probe depth of MEDRANK vs. the probe depth and the fraction of database accessed by L2TA, as a function of dimension. It is easy to see that L2TA accesses a large constant fraction of the database,[7] whereas MEDRANK accesses only a tiny fraction.

4. CONCLUSIONS

We have introduced rank aggregation as a new approach towards doing similarity search and classification. We take the query and the candidates to be points in a multidimensional space. Each coordinate is treated as a voter, who ranks the points based on closeness to the corresponding coordinate of the query. The winners are those points with the highest aggregated ranks. Combined with dimensionality reduction, this approach yields a simple, database-friendly algorithm that gives a very good approximate answer to the nearest neighbor problem. The algorithm is extremely efficient, often exploring no more than 5% of the data to obtain very high-quality results. We feel that the approach is conceptually interesting in its own right, not just as an approximation to nearest neighbors. Our results also highlight median rank aggregation as an efficient and useful form of rank aggregation.

An interesting research direction is to consider the case of small domains where several points are forced to have the same rank and to to adapt our methodology to exploit this feature.

[7]This seems to be because of the high dimensionality of the data in all of our experiments.

5. REFERENCES

[1] C. Aggarwal. Hierarchical subspace sampling: A unified framework for high dimensional data reduction, selectivity estimation, and nearest neighbor search. In *Proceedings of the ACM SIGMOD Conference*, pages 452–463, 2002.

[2] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.

[3] J. J. Bartholdi, C. A. Tovey, and M. A. Trick. The computational difficulty of manipulating an election. *Social Choice and Welfare*, 6(3):227–241, 1989.

[4] S. Berchtold, D. Keim, and H.-P. Kriegel. The X-Tree: An index structure for high dimensional data. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, 1996.

[5] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbors meaningful? In *Proceedings of the 7th International Conference on Database Theory*, pages 217–235, 1999.

[6] M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 380–388, 2002.

[7] P. Diaconis and R. Graham. Spearman's footrule as a measure of disarray. *Journal of the Royal Statistical Society, Series B*, 39(2):262–268, 1977.

[8] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proceedings of the 10th International World Wide Web Conference*, pages 613–622, 2001.

[9] R. Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences*, 58:83–99, 1999.

[10] R. Fagin. Combining fuzzy information: an overview. *ACM SIGMOD Record*, 31(2):109–118, 2002.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the 20th ACM Symposium on Principles of Database Systems*, pages 102–113, 2001.

[12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Databases*, pages 518–529, 1999.

[13] J. Goldstein and R. Ramakrishnan. Contrast plots and P-sphere trees: space vs. time in nearest neighbour searches. In *Proceedings of the 28th International Conference on Very Large Databases*, pages 429–440, 2002.

[14] E. Hemaspaandra, L. A. Hemaspaandra, and J. Rothe. Exact analysis of Dodgson elections: Lewis Carroll's 1876 voting system is complete for parallel access to NP. In *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming*, pages 214–224, 1997.

[15] I. Ilyas, W. Aref, and A. Elmagarmid. Joining ranked inputs in practice. In *Proceedings of the 28th International Conference on Very Large Databases*, pages 950–961, 2002.

[16] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[17] J. G. Kemeny. Mathematics without numbers. *Daedalus*, 88:571–591, 1959.

[18] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 599–608, 1997.

[19] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM Journal on Computing*, 30(2):451–474, 2000.

[20] D. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, 1996.

[21] H. P. Young. Condorcet's theory of voting. *American Political Science Review*, 82:1231–1244, 1988.

[22] H. P. Young and A. Levenglick. A consistent extension of Condorcet's election principle. *SIAM Journal on Applied Mathematics*, 35(2):285–300, 1978.