KDX: An Indexer for Support Vector Machines

Navneet Panda and Edward Y. Chang, *Senior Member, IEEE*

Abstract—Support Vector Machines (SVMs) have been adopted by many data mining and information-retrieval applications for learning a mining or query concept, and then retrieving the "top-k" best matches to the concept. However, when the data set is large, naively scanning the entire data set to find the top matches is not scalable. In this work, we propose a kernel indexing strategy to substantially prune the search space and, thus, improve the performance of top-k queries. Our *kernel indexer* (KDX) takes advantage of the underlying geometric properties and quickly converges on an approximate set of top-k instances of interest. More importantly, once the kernel (e.g., Gaussian kernel) has been selected and the indexer has been constructed, the indexer can work with different kernel-parameter settings (e.g., γ and σ) without performance compromise. Through theoretical analysis and empirical studies on a wide variety of data sets, we demonstrate KDX to be very effective. An earlier version of this paper appeared in the 2005 SIAM International Conference on Data Mining [24]. This version differs from the previous submission in providing a detailed cost analysis under different scenarios, specifically designed to meet the varying needs of accuracy, speed, and space requirements, developing an approach for insertion and deletion of instances, presenting the specific computations as well as the geometric properties used in performing the same, and providing detailed algorithms for each of the operations necessary to create and use the index structure.

Index Terms—Support vector machine, indexing, top-k retrieval.

1 Introduction

Support Vector Machines (SVMs) [7], [29] have become increasingly popular over the last decade because of their superlative performance and wide applicability. SVMs have been successfully used for many data mining and information-retrieval tasks, such as outlier detection [1], classification [5], [17], [19], novelty detection [27], and query-concept formulation [9], [28]. SVMs have been successfully used for handwritten digit recognition [12], [26], [6], face detection in images [23], text categorization [17], [13], [11], protein and gene classification [20], [25] and cancer tissue classification [14], [16]. In these applications, SVMs learn a prediction function as a hyperplane to separate the training instances relevant to the target concept (representing a pattern or a query) from the others. The hyperplane is depicted by a subset of the training instances called *support vectors*. The unlabeled instances are then given a score based on their distances to the hyperplane. Many data mining and information-retrieval tasks query for the "top-k" best matches to a target concept. Yet, it would be naive to require a linear scan of the entire unlabeled pool, which may contain thousands or millions of instances, to search for the top-k matches. To avoid a linear scan, we propose a kernel indexer (KDX) to work with SVMs. We demonstrate its scalable performance for top-k queries.

Traditional top-k query scenarios use a point in a vector space to depict the query, so the top-k matches are the k nearest instances to the query point in the vector space. A top-k query with SVMs differs from that in the traditional scenarios in two aspects. First, a query concept learned by SVMs is represented by a hyperplane, not by a point. Second, a top-k query with SVMs can request the farthest instances from the hyperplane (the top-k matches for a concept), or those nearest to it (the top-k uncertain instances[1] for a concept). KDX supports top-k match to a query concept.

Intuitively, KDX works as follows: Given a kernel function and an unlabeled pool, KDX first finds the approximate center instance of the pool in the feature space. It then divides the feature space, to which the kernel function projects the unlabeled instances, into concentric hyperrings (hereafter referred to as *rings* for brevity). Each ring contains about the same number of instances and is populated by instances according to their distances to the center instance in the feature space. Given a query concept, represented by a hyperplane, KDX limits the number of rings examined, and intelligently prunes out unfit instances from each ring. Finally, KDX returns the top-k results. Both the *inter-ring pruning* and *intra-ring pruning* are performed by exploiting the geometric properties of the feature space. (Details are presented in Section 4.)

KDX supports three important properties. First, it can effectively support insertion and deletion operations. Second, given a kernel function, the index can be constructed independent of the settings of the kernel parameters (e.g., γ and σ). This parameter-invariant property is especially crucial, since varied query-concepts can best be learned under variable parameter settings and rebuilding the index each time the parameter is changed is not feasible. Third, KDX allows us to maintain sequential disc access, thus allowing fast retrieval of the associated index structure for all computations. Through empirical studies on a wide

• N. Panda is with the Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106. E-mail: panda@cs.ucsb.edu.
• E.Y. Chang is with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106. E-mail: echang@ece.ucsb.edu.

Manuscript received 17 Mar. 2005; revised 11 Sept. 2005; accepted 7 Nov. 2005; published online 20 Apr. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0096-0305.

1. In an active learning setting, the algorithm finds the most uncertain instances to query the user for labels. The most uncertain instances are the ones closest to the hyperplane.

variety of data sets, we demonstrate KDX to be very effective.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 provides an overview on SVMs and introduces geometric properties useful to our work. We then propose KDX in Section 4, describing its key operations: index creation, top-k farthest instances lookup, and updates. Section 5 presents the results of our empirical studies. We offer our concluding remarks in Section 6, together with suggestions for future research directions.

2 Related Work

Indexing for SVMs to support top-k queries can be very challenging for three reasons. First, a kernel function K is the dot product of a basis function Φ, but we may not explicitly know the basis functions of most kernels. Second, even if the basis function is known, the dimension of the feature space F, to which the instances are projected, can be very high, possibly infinite. It is well known that traditional indexing methods do not work well with high-dimensional data for nearest-neighbor queries [30]. Third, a query represented by SVMs is a hyperplane, not a point.

Indexing has been intensively studied over the past few decades. Existing indexers can be divided into two categories: *coordinate-based* and *distance-based*. The coordinate-based methods work on objects residing in a vector space by partitioning the space. A top-k query can be treated as a range query and, ideally, only a small number of partitions need to be scanned for finding the best matches. Example coordinate-based methods are the X-tree [3], the R^*-tree [2], the TV-tree [22], and the SR-tree [18], to name a few. All these indexers need an explicit feature representation to be able to partition the space. As discussed above, the feature space onto which an SVM kernel projects data might not have an explicit representation. Even in cases where the projection function Φ is known, the dimension of the projected space could be too high to use the coordinate-based methods due to the curse of dimensionality [21]. Thus, the traditional coordinate-based methods are not suitable for kernel indexing.

Distance-based methods do not require an explicit vector space. The M-tree [10] is a representative scheme that uses the distances between instances to build an indexing structure. Given a query point, it prunes out instances based on their distances. SVMs use the distance from the hyperplane as a measure of the suitability of an instance. The farther the instance from the hyperplane in the positive half-space, the higher its "score" or confidence. The traditional distance-based methods require a query to be a *point*, whereas in this case, we have a *hyperplane*. With infinite number of points on the query hyperplane, a top-k query using points may require scanning all buckets of the index.

When the data dimension is very high, the cost of supporting exact queries can be higher than that of a linear scan. The work of [15] proposes an approximate indexing strategy using latent semantic hashing. This approach hashes similar instances into the same bucket with a high degree of accuracy. A top-k approximate query can be supported by retrieving the bucket into which the query point has been hashed. Unfortunately, this method requires the knowledge of the feature vector in the projected space, and cannot be used with SVMs. Another approximate approach is clustering for indexing [21], but this approach supports only point-based queries, not hyperplane queries.

3 Preliminaries

We briefly present SVMs, and then discuss the geometrical properties that are useful in the development of the proposed indexing structure.

3.1 Support Vector Machines

Let us consider SVMs in the binary classification setting. We are given a set of data $\{\mathbf{x}_1, \ldots, \mathbf{x}_{m+n}\}$ that are vectors in some space $\mathsf{X} \subseteq \mathbb{R}^d$. Among the $m+n$ instances, m of them, denoted as $\{\mathbf{x}_{l,1}, \ldots, \mathbf{x}_{l,m}\}$, are assigned labels $\{y_1, \ldots, y_m\}$, where $y_i \in \{-1, 1\}$. The rest are unlabeled data, denoted as $\{\mathbf{x}_{u,1}, \ldots, \mathbf{x}_{u,n}\}$. The labeled instances are also called training data; and unlabeled are sometimes called testing data. In the remainder of this paper, we refer to a training instance simply as $\mathbf{x}_{l,i}$, and a testing instance as $\mathbf{x}_{u,i}$. When we just refer to an instance, either training or testing, we use $\mathbf{x}_i$.

In the simplest form, SVMs are hyperplanes that separate the training data by a maximal margin. A hyperplane separates the training data such that all vectors lying on one side of the hyperplane are labeled as -1, and all vectors lying on the other side are labeled as 1. The training instances that lie closest to the hyperplane are called *support vectors*. SVMs allow us to project the original training data in space X to a higher-dimensional feature space F via a Mercer kernel operator K. Thus, by using K, we implicitly project the training data into a different (often higher-dimensional) feature space F.

The SVM computes the weights $\{\alpha_i\}$ associated with the training instances that correspond to the maximal margin hyperplane in F. By choosing various kernel functions (discussed shortly), we can implicitly project the training data from X into various feature spaces. (A hyperplane in F maps to a more complex nonlinear decision boundary in the original space X.) Once the hyperplane has been learned based on the training data $\{\mathbf{x}_{l,1} \ldots \mathbf{x}_{l,m}\}$, the class membership of an unlabeled instance $\mathbf{x}_{u,r}$ can be predicted using the α_is of the training instances and their labels $\{y_1, \ldots, y_m\}$ by

$$f(\mathbf{x}_{u,r}) = \sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_{l,i}, \mathbf{x}_{u,r}) + b, \qquad (1)$$

b being the displacement of the hyperplane from the origin. When $f(\mathbf{x}_{u,r}) \geq 0$, we classify $\mathbf{x}_{u,r}$ as $+1$; otherwise, we classify $\mathbf{x}_{u,r}$ as -1.

SVMs rely on the values of inner products between pairs of instances to measure their similarity. The kernel function K computes the inner products between instances in the feature space. Mathematically, a kernel function can be written as,

$$K(\mathbf{x}_1, \mathbf{x}_2) = <\phi(\mathbf{x}_1), \phi(\mathbf{x}_2)>, \qquad (2)$$

where ϕ is the implicit mapping used for projecting the instances, $\mathbf{x}_1$ and $\mathbf{x}_2$. Essentially, the kernel function takes as input, a pair of instances, and returns the similarity between them in the feature space. Commonly used kernel functions are the Gaussian, the Laplacian kernels, and the Polynomial. These are expressed as:

1. Gaussian : $K(\mathbf{x}_1, \mathbf{x}_2) = exp(\frac{-\|\mathbf{x}_1 - \mathbf{x}_2\|_2^2}{2\sigma^2})$.
2. Laplacian : $K(\mathbf{x}_1, \mathbf{x}_2) = exp(-\gamma \parallel \mathbf{x}_1 - \mathbf{x}_2 \parallel_1)$.
3. Polynomial : $K(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1 \cdot \mathbf{x}_2 + 1)^p$.

The tunable parameters, σ for Gaussian, γ for the Laplacian kernel, and p for Polynomial, define different mappings. In each of the above, the mapping function ϕ is not defined explicitly. Yet, the inner product in the feature space can be evaluated in terms of the input space vectors and the corresponding parameter (σ, γ, or p) for the chosen kernel function.

3.2 Geometrical Properties of SVMs

We present three geometrical properties of kernel-based methods used extensively throughout the rest of the paper.

1. *Similarity between any two instances measured by a kernel function is between zero and one.* Commonly used kernels like the Gaussian and the Laplacian are normalized kernels where the similarity between instances, as measured by the kernel function, takes on values between 0 and 1. A value of 1 indicates that the instances are identical whereas a value of 0 means they are completely dissimilar. The polynomial kernel, though not necessarily normalized, can easily be normalized by using
$$K_n(\mathbf{x}_1, \mathbf{x}_2) = \frac{K(\mathbf{x}_1, \mathbf{x}_2)}{\sqrt{K(\mathbf{x}_1, \mathbf{x}_1)K(\mathbf{x}_2, \mathbf{x}_2)}}, \quad (3)$$
where K_n is the normalized kernel function. Here, we have assumed that the features associated with the instances are positive or zero. If not, appropriate scaling of the features can be performed. Normalization is a prerequisite and our method only works with normalized kernels. Please note that normalizing the polynomial kernels does in fact change the feature space and the hyperplane learned in the normalized space in the case of polynomial kernels is not in general the same as the hyperplane learned for the unnormalized kernel.
2. *The projected instances lie on the surface of a unit hypersphere.* For a normalized kernel, the inner product of an instance with itself, $K_n(\mathbf{x}_i, \mathbf{x}_i)$, is equal to 1. This means that, after projection, all the instances lie on the surface of a hypersphere. Further, considering the fact that the kernel values are inner products, we see that the relative angle[2] in feature space between any two instances is bounded above by $\frac{\pi}{2}$. This is so since the inner product is constrained to be always greater than or equal to $0(cos^{-1}(0) = \frac{\pi}{2})$. Henceforth, we refer to the relative angle between instances as the angle between them.
3. *Data instances exist on both sides of a query hyperplane.* The hyperplane needs to pass through the region on the hypersphere populated by the projected instances. Otherwise, it would be impossible to separate the positive from the negative training samples. This property is easily ensured since we have at least one training instance from the positive class and one from the negative class.

2. Relative angles constrain the interinstance angle to be within π.

4 KDX

In this section, we present our indexing strategy, KDX, for finding the top-k *relevant* instances. We discuss the construction of the index in Section 4.1, the approach for finding the top-k instances in Section 4.2, handling changes in kernel parameters in Section 4.3, and insertion and deletion operations in Section 4.4.

Definition 1. Top-k *Relevant Instances. Given the set of instances $S = \{\mathbf{x}_r\}$, and the normal to the hyperplane, $\mathbf{w}$, represented in terms of the the support vectors, the* top-k *relevant instances are the set of instances $(\mathbf{q}_1, \mathbf{q}_2, \cdots, \mathbf{q}_k) \subset S$ such that $\sum_{i=1,\mathbf{q}_i \in S}^{k} \mathbf{w} \cdot \phi(\mathbf{q}_i)$ is* maximized *over all possible choices of $\mathbf{q}_1, \cdots, \mathbf{q}_k$ with $\mathbf{q}_i \neq \mathbf{q}_j$ if $i \neq j$. The subscripts do not represent the order of their membership in S. Ties are broken arbitrarily.*

4.1 KDX-create

The indexer is created in four steps:

1. finding the instance $\phi(\mathbf{x}_c)$ that is approximately centrally located in the feature space F,
2. separating the instances into rings based on their angular distances from the central instance $\phi(\mathbf{x}_c)$,
3. constructing a local indexing structure (*intra-ring indexer*) for each ring, and
4. creating an *inter-ring* index.

The index thus consists of two parts, an intra-ring index and an inter-ring index. The purpose of each of these parts is explained in detail in Section 4.2. The intra-ring index is used for the actual pruning while the inter-ring index is used to provide good starting points for the algorithm. The formal algorithm is presented in Fig. 2.

4.1.1 Finding the Central Instance

As shown in Fig. 1, we attempt to find an approximate center $\phi(\mathbf{x}_c)$ after the implicit projection of the instances to the feature space F by kernel function K. The cosine of the angle between a pair of instances is given by the value of the kernel function K with the two instances as input (see (2)).

Lemma 2. *The closest approximation of the central instance is the instance $\mathbf{x}_c$ whose sum of distances from the other instances is the smallest.*

Proof. The point in F whose coordinates are the average of the coordinates of the projected instances in the data set is at the center of the distribution of instances $\phi(\mathbf{x}_i)$, $i = 1 \dots n$. Choosing the instance which minimizes the variance gives us the closest approximation to the true center since it is closest to the point with average coordinates in F.

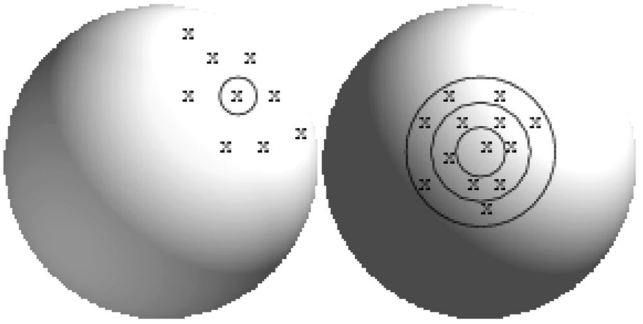
Fig. 1. Approximate central instance and rings.

$$\mathbf{x}_c = argmin_{\mathbf{x}_j} \sum_i \parallel \phi(\mathbf{x}_i) - \phi(\mathbf{x}_j) \parallel^2$$
$$= argmin_{\mathbf{x}_j} \sum_i (2 - 2K(\mathbf{x}_i, \mathbf{x}_j)).$$

□

Given n instances in the data set, each with d features, finding the central instance in the projected space takes $O(n^2 d)$ time.[3] This step can be achieved with $O(1)$ storage because at any point we need to store just the current known minimum, and the accumulated value of the sum of the angles of the rest of the instances with the current instance.

4.1.2 Separating Instances into Rings

In this step, we compute the angles of the projected instances in F with the central instance, $\phi(\mathbf{x}_c)$, using K. The angles are stored in an array, which is then sorted. To divide the instances into rings, we equally divide the sorted list. That is, if the number of instances per ring is g, then the first g elements in the sorted array are grouped together to form the first ring, and so on. Formally, an instance at the rth position in the sorted list is part of the $\lfloor r/g \rfloor$th ring.

Here, we have a choice of the number of instances that need to be included in a ring. The number of instances per ring can be based on the size of the L2 cache on the system to minimize cache misses. As we shall see later, only the instances in the same ring are processed together. Hence, at any given time during the processing of queries, we need only the amount of storage utilized by the instances in one ring. Fig. 1 shows the division of instances into different rings. This step requires $O(n \log n)$ time, and $O(n)$ space.

4.1.3 Constructing Intra-Ring Index

For each ring, KDX constructs a local index. We construct for each ring a $g \times g$ square matrix, where the ith row of the matrix contains the angles between the ith instance and the other $g - 1$ instances. Next, we sort each row such that the instances are arranged according to decreasing order of similarity (or increasing order of distance) with the instance associated with the row.

This step requires $O(g^2)$ storage and $O(g^2\ d) + O(g^2\ log\ g)$ computational time for each ring.

3. Since we are only interested in the approximate central instance this cost can be easily lowered via a sampling method.

4.1.4 Creating Inter-Ring Index

Finally, we construct the inter-ring index, which is the closest instance from the adjoining ring for each instance. This step requires $O(n)$ storage and $O(ng)$ time. All the steps above are essentially preprocessing of the data which needs to be done only once for the data set.

The overall computational cost of creating the index is given by $O(n^2\ d)$ assuming $n\ d > g\ log\ g$. The storage requirement assuming that the intra-ring index is stored in memory is given by $O(n\ g)$. In case the rings index is computed just in time, the storage requirement is reduced to $O(n\ d)$, which is essentially the same as the space required for all the instances.

4.2 KDX-top_*k*

In this section, we describe how KDX finds top-k instances relevant to a query (Definition 1) by just examining a fraction of the data set. Let us revisit Definition 1 for top-k relevant queries. The most relevant instances to a query, represented by a hyperplane trained by SVMs, are the ones farthest from the hyperplane on the positive side. Without an indexer, finding the farthest instances involves computing the distances of all the instances in the data set from the hyperplane, and then selecting the k instances with greatest distances. This linear-scan approach is clearly costly when the data set is large. Further, the number of dimensions associated with each data instance has a multiplicative effect on this cost. KDX performs inter-ring and intra-ring pruning to find the approximate set of top-k instances by:

1. Shifting the hyperplane to the origin parallel to itself, and then computing θ_c, the angular distance between the normal to the hyperplane and the central instance $\phi(\mathbf{x}_c)$.
2. Identifying the ring with the farthest coordinate from the hyperplane, and selecting a starting instance $\phi(\mathbf{x})$ in that ring.
3. Computing the angular separation between $\phi(\mathbf{x})$ and the farthest coordinate in the ring from the hyperplane, denoted as $\phi(\mathbf{x}^*)$.
4. Iteratively, replacing $\phi(\mathbf{x})$ with a closer instance to $\phi(\mathbf{x}^*)$ and updating the top-k list, until no "better" $\phi(\mathbf{x})$ in the ring can be found.
5. Identifying a good starting instance $\phi(\mathbf{x})$ for the next ring, followed by repeating steps 3 to 5, until the termination criterion is satisfied.

KDX achieves speedup over the naive linear scan method in two ways. First, KDX does not examine all rings for a query. KDX terminates its search for top-k when the constituents of the top-k set do not change over the evaluation of multiple rings, or the query time expires. Second, in the fourth step, KDX examines only a small fraction of the instances in a ring. The remainder of this section details these steps, explaining how KDX effectively approximates the top-k[4] result for achieving significant speedup. The formal algorithm is presented in Fig. 11 and Fig. 12.

4. In the event that less than k relevant instances exist in the data set, the approach returns only the p instances ($p < k$) classified as positive by the hyperplane.

4.2.1 Computing θ_c

We compute the angular distance θ_c of the central instance $\phi(\mathbf{x}_c)$ from the normal to the hyperplane. Parameter θ_c is important for KDX to identify the ring containing the farthest coordinate from the hyperplane. To compute θ_c, we first shift the hyperplane to pass through the origin in the feature space. The SVM training phase learns the distance of the hyperplane from the origin in terms of variables b and $\mathbf{w}$ [29]. The distance of the hyperplane from the origin is given by $-b/\|\mathbf{w}\|$. We shift the hyperplane to pass through the origin without changing its orientation by setting $b = 0$. This shift does not affect the set of instances farthest from the hyperplane because it has the same effect as adding a constant value to all distances.

Given training instances $\mathbf{x}_{l,1} \ldots \mathbf{x}_{l,m}$ and their labels $y_1 \ldots y_m$, SVMs solve for weights α_i for $\mathbf{x}_{l,i}$. The normal of the hyperplane[5] can be written as

$$\mathbf{w} = \frac{\sum_i^m \alpha_i y_i \phi(\mathbf{x}_{l,i})}{\sqrt{\sum_{i,j}^m \alpha_i \alpha_j y_i y_j \phi(\mathbf{x}_{l,i}) \cdot \phi(\mathbf{x}_{l,j})}}. \tag{4}$$

The angular distance between the central instance and $\mathbf{w}$ is essentially $cos^{-1}(\mathbf{w} \cdot \phi(\mathbf{x}_c))$.[6]

4.2.2 Identifying the Starting Ring

The most logical ring from which to start looking for the farthest instance is the one containing the coordinate on the hypersphere farthest from the hyperplane. Let $\phi(\mathbf{x}^\diamond)$ denote this farthest coordinate. Note that there may not exist a data instance at $\phi(\mathbf{x}^\diamond)$. However, finding an instance close to the farthest coordinate can help us find the farthest instance with high probability. The following lemma shows how we can identify the ring containing the farthest coordinate from the hyperplane.

Lemma 3. *The point, $\phi(\mathbf{x}^\diamond)$, on the surface of the hypersphere, farthest from the hyperplane, is at the intersection of the hypersphere and the normal to the hyperplane passing through the origin.*

The proof follows from the fact that all the instances are constrained to lie on the surface of a hypersphere and the distance from the hyperplane decreases as we move away from the point of intersection of the normal with the hypersphere because of the curvature.

We do not need to explicitly compute the farthest coordinate, since we are only interested in the ring where it resides. To find the ring, we rely on the angular separation of $\phi(\mathbf{x}^\diamond)$ from $\phi(\mathbf{x}_c)$, which is the θ_c obtained in the previous section. We use Fig. 3 to illustrate. The figure shows that $\phi(\mathbf{x}^\diamond)$ is at the intersection of the hypersphere and the normal to the hyperplane with θ_c angular separation from $\phi(\mathbf{x}_c)$. Given $\mathbf{x}_c$ and the normal of the hyperplane, we can compute θ_c to locate the ring containing the farthest coordinate on the hypersphere from the hyperplane. The rings were formed from the sorted array of instances based on their angular separation from the central instance. Therefore, the first instance picked for every ring serves as a delimiter for that ring. To identify the ring, we therefore need to look only at these delimiters.

4.2.3 Intra-Ring Pruning

Our goal is to find the farthest instances in the ring from the hyperplane. In this section, we present our pruning algorithm, which aims to reduce the number of instances examined to find a list of *approximate* farthest instances. In Section 5, we show that our pruning algorithm achieves high-quality top-k results, just by examining a small fraction of instances.

If the ring is the first one being evaluated, KDX randomly chooses an instance $\phi(\mathbf{x})$ in the ring as the anchor instance. (In Section 4.2.6, we show that if the ring is not the first to be inspected, we can take advantage of the inter-ring index to find a good $\phi(\mathbf{x})$.) Let $\phi(\mathbf{x}^*)$ be the farthest point from the hyperplane in the ring. We would like to find instances in the ring closest to $\phi(\mathbf{x}^*)$. Our goal is to find these instances by inspecting as few instances in the ring as possible.

Let us use a couple of figures to illustrate how this intra-ring pruning algorithm works. First, the circle in Fig. 4 depicts the hyperdisc of the current ring. Please note that the hyperdisc can be inclined at an angle to the hyperplane as shown in Fig. 5. Back to Fig. 4. We would like to compute the distance s between $\phi(\mathbf{x})$ and $\phi(\mathbf{x}^*)$. Since both $\phi(\mathbf{x})$ and $\phi(\mathbf{x}^*)$ lie on the surface of a unit hypersphere, the angular separation between them can be obtained once s is known. Fig. 4 shows that we need to determine h and v in order to use the Pythagorus theorem to obtain s. Determination of h and v, in turn, requires the knowledge of distances d_1 and d_2. Distance d_1 denotes the distance from the center of the hyperdisc to the hyperplane, along the hyperdisc, and d_2 the distance of $\phi(\mathbf{x})$ to the hyperplane, along the hyperdisc. It is noteworthy that both these distances are measured along the surface of the hyperdisc as shown for d_2 in Fig. 5.

We now discuss the geometrical aspects of the proposed method and demonstrate how the various values (d_1, d_2, and r) in Fig. 4 are computed. Essentially, the rings can be visualized as in Fig. 3. Visualizing the situation from a direction parallel to the surface of the rings gives us Fig. 4. Here, we have shown the hyperplane passing through the origin but this is not necessary for our method since a hyperplane which does not pass through the origin can always be shifted parallel to itself to the origin without changing the instances which are farthest away from it in the positive half-space. The only situation where we would not be able to do this is when the hyperplane was such that all the instances were only in the negative half-space, but then such a hyperplane is useless.

4.2.4 Computation of ψ

Observing Fig. 3, we note that the rings formed by each of the rings are inclined at the same angle to the hyperplane (and, hence, its normal) The angle of inclination can be found by computing the angular separation of the central instance with the normal. If the central instance makes an angle greater than $\pi/2$ with the normal and its angular separation from the normal is θ_c, then the angle of inclination ψ is given by

5. Training instances with zero weights are not support vectors and do not affect the computation of the normal.

6. From a computational perspective, it is important to note that the $\mathbf{w} \cdot \phi(\mathbf{x})$ takes values in a narrow range rather than over the whole possible range $\{-1, 1\}$. Thus, even the instance farthest from the hyperplane has a small value of the inner product with $\mathbf{w}$. To handle such issues, we scale the inner products of instances with $\mathbf{w}$ to take values between $\{-1, 1\}$. The scaling is performed based on the highest training score, t_{max}. Thus, the range $\{-1, t_{max}\}$ is scaled to $\{-1, 1\}$.

```
1: Input:  Dataset instances $\mathbf{x}_i$
           Number of instances per ring $g$
   Output: Central instance $\mathbf{x}_c$
           Intra-ring index $Arr$
           Inter-ring index $inter\_ring$
           Ring entries $ring$
2: $\mathbf{x}_c$ =Find_central($\mathbf{x}_i$)
3: $ring$ = Separate_instances($\mathbf{x}_i, \mathbf{x}_c, g$)
4: $Arr$ = Create_intra_ring_index($ring, g, \mathbf{x}_i$)
5: $inter\_ring$ = Create_inter_ring_index($ring, \mathbf{x}_i, g$)
```

```
Procedure Find_central($\mathbf{x}_i$)

1: $n$ = Number of instances in dataset
2: $min = \inf$
3: $c$ /* Index of central instance */
4: for $i = 1$ to $n$ do
5:   sum[i] $= 0$
6:   for $j = 1$ to $n$ do
7:     sum[i] $+= (2 - 2K(\mathbf{x}_i, \mathbf{x}_j))$
8:   end for
9:   if $min >$ sum[i] then
10:     $min$ = sum[i]
11:     $c = i$
12:   end if
13: end for
```

```
Procedure Separate_instances($\mathbf{x}_i, g, \mathbf{x}_c$)

1: for $i = 1$ to $n$ do
2:   $angle_c[i] = K(\mathbf{x}_i, \mathbf{x}_c)$
3:   $index_c[i] = i$
4: end for
5: Sort( $angle_c, index_c$ ) /* $index_c$ stores the instance number after sorting*/
6: $r = 1$
7: for $j = 1$ to $n$ do
8:   if $j\%g == 0$ then
9:     $r = r + 1$
10:   end if
11:   $ring[r][j\%g + 1] = index_c[j]$
12:   $inverse\_index[index_c[j]] = j\%g + 1$ /*Stores the position in the ring*/
13: end for
```

```
Procedure Create_intra_ring_index($ring, \mathbf{x}_i, g$)

1: $g$ = Number of instances in each ring
2: for $i = 1$ to $n/g$ /*Cycle over all rings*/ do
3:   for $j = 1$ to $g$ do
4:     for $k = 1$ to $g$ do
5:       $Arr[i][j][k] = K(\mathbf{x}_{ring[i][j]}, \mathbf{x}_{ring[i][k]})$
6:     end for
7:     Sort( $Arr[i][j]$ )
8:   end for
9: end for
```

```
Procedure Create_inter_ring_index($ring, \mathbf{x}_i, g$)

1: for $i = 1$ to $n/g - 1$ do
2:   for $j = 1$ to $g$ do
3:     $max = 0$
4:     for $k = 1$ to $g$ do
5:       if $K(\mathbf{x}_{ring[i][j]}, \mathbf{x}_{ring[i+1][k]}) > max$ then
6:         $max = K(\mathbf{x}_{ring[i][j]}, \mathbf{x}_{ring[i+1][k]})$
7:         $inter\_ring[ring[i][j]][0] = ring[i+1][k]$
8:       end if
9:     end for
10:   end for
11:   for $j = 1$ to $g$ do
12:     $max = 0$
13:     for $k = 1$ to $g$ do
14:       if $K(\mathbf{x}_{ring[i][j]}, \mathbf{x}_{ring[i-1][k]}) > max$ then
15:         $max = K(\mathbf{x}_{ring[i][j]}, \mathbf{x}_{ring[i-1][k]})$
16:         $inter\_ring[ring[i][j]][1] = ring[i-1][k]$
17:       end if
18:     end for
19:   end for
20: end for
```

Fig. 2. Algorithm for creating the index.

$$\psi = \pi - \theta_c. \tag{5}$$

If the central instance makes an angle of less than $\pi/2$ with the normal to the hyperplane then the angle of inclination is given by

$$\psi = \theta_c. \tag{6}$$

Both the situations are presented in Fig. 7.

4.2.5 *Computation of Distances d_1 and d_2*

Once the angle of inclination has been determined we can find the distance of $\phi(\mathbf{x})$ from the intersection of the hyperplane with the surface of the ring. We know the distance of $\phi(\mathbf{x})$ from the hyperplane. In Fig. 5, this is represented by d. Knowing d and the angle of inclination allows us to compute d_2.

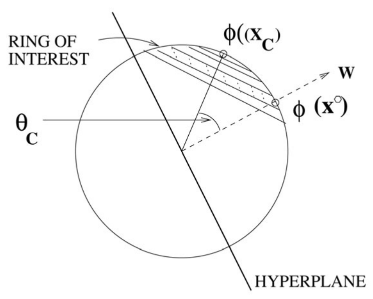

Fig. 3. Start ring.

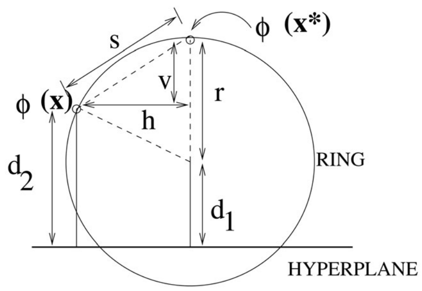

Fig. 4. Finding s.

The radius of the ring, r, in Fig. 8 can be computed since we know the radius of the hypersphere (1) and the angle between $\phi(\mathbf{x})$ and $\phi(\mathbf{x}_c)$. Similarly, we can determine p which is the distance of the center of the ring from the origin O. Once p has been determined, the distance of the center from the intersection of the hyperplane and the disc (d_1) can be directly computed since we know the angle of inclination of the disc to the hyperplane. Thus, we have $d_1 = p/tan(\psi)$.

Given $\phi(\mathbf{x})$ and s, KDX at each step tries to find an instance farther than $\phi(\mathbf{x})$ from the hyperplane and closer to $\phi(\mathbf{x}^*)$. Such an instance would lie between $\phi(\mathbf{x}^*)$ and $\phi(\mathbf{x})$, or between $\phi(\mathbf{x}^*)$ and point C, as depicted in Fig. 6. Once we find a "better" instance than $\phi(\mathbf{x})$, we replace $\phi(\mathbf{x})$ with the new instance, and search for yet another farther instance. Notice that as we find a farther $\phi(\mathbf{x})$ from the hyperplane, the search range between $\phi(\mathbf{x})$ and C is reduced. This pruning algorithm eventually converges when no instances reside in the search range. When the pruning algorithm converges, there is a high probability that we have found a point $\phi(\mathbf{x})$ in the ring that is the farthest from the hyperplane.

To understand the computational savings of this intra-ring pruning algorithm, let us move down to the next level of details. We use the example in Fig. 9 to explain the pruning process. Starting at $\phi(\mathbf{x})$, we seek to find an instance as close to $\phi(\mathbf{x}^*)$ as possible. The intra-ring index (Section 4.1.3) of

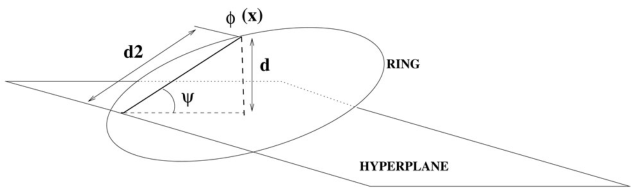

Fig. 5. Distance of $\phi(\mathbf{x})$ from intersection of hyperplane and disc.

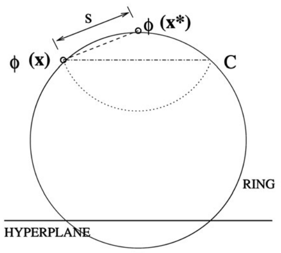

Fig. 6. Stopping condition.

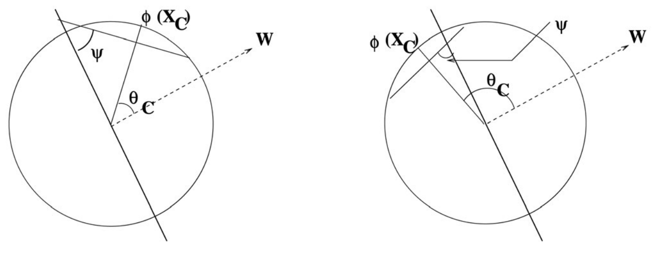

Fig. 7. Finding the angle of inclination.

$\phi(\mathbf{x})$ contains an ordered list of instances based on their distances from $\phi(\mathbf{x})$. Let τ denote the angular separation between $\phi(\mathbf{x})$ and $\phi(\mathbf{x}^*)$. To find an instance close to $\phi(\mathbf{x}^*)$, we search this list for instances with an angular separation of about τ from $\phi(\mathbf{x})$. For the example in Fig. 9, the neighboring points of $\phi(\mathbf{x})$ appear in the order $\phi(\mathbf{x}_3)$, $\phi(\mathbf{x}_1)$, $\phi(\mathbf{x}_4)$, $\phi(\mathbf{x}_5)$, $\phi(\mathbf{x}_2)$, $\phi(\mathbf{x}_6)$, $\phi(\mathbf{x}_7)$, and $\phi(\mathbf{x}_8)$ in the sorted list of $\phi(\mathbf{x})$. First, we need only examine the instances lying within the arc PQ in the figure, since an instance outside this arc cannot be closer to $\phi(\mathbf{x}^*)$ than $\phi(\mathbf{x})$ itself. This step allows us to prune instances $\phi(\mathbf{x}_8)$ and $\phi(\mathbf{x}_7)$.

Next, we would like to resort the instances remaining on the list of $\phi(\mathbf{x})$ based on their likelihood of being close to $\phi(\mathbf{x}^*)$. To quantify this likelihood for instance $\phi(\mathbf{x}_i)$, we compute how close the angular distance between $\phi(\mathbf{x}_i)$ and $\phi(\mathbf{x})$ is to the angular distance between $\phi(\mathbf{x}^*)$ and $\phi(\mathbf{x})$ (which is τ). The list does not need to be explicitly constructed since we have sorted and stored the distances between $\phi(\mathbf{x}_i)$ and $\phi(\mathbf{x})$ in the intra-ring index. Once we find the instance closest to $\phi(\mathbf{x}^*)$ in the index, the rest of the instances on the resorted list can be obtained by looking up the adjacent instances of the closest instance in the intra-ring index. In our example, this resorted list is $\phi(\mathbf{x}_4)$, $\phi(\mathbf{x}_5)$, $\phi(\mathbf{x}_1)$, $\phi(\mathbf{x}_3)$, $\phi(\mathbf{x}_2)$, and $\phi(\mathbf{x}_6)$.

It may be surprising that $\phi(\mathbf{x}_5)$ and $\phi(\mathbf{x}_4)$ appear before $\phi(\mathbf{x}_1)$ on the resorted list. The reason is that we know only

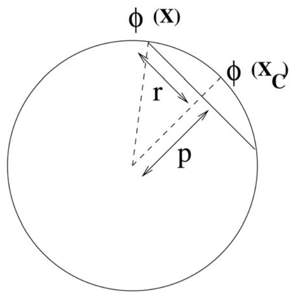

Fig. 8. Determination of the radius.

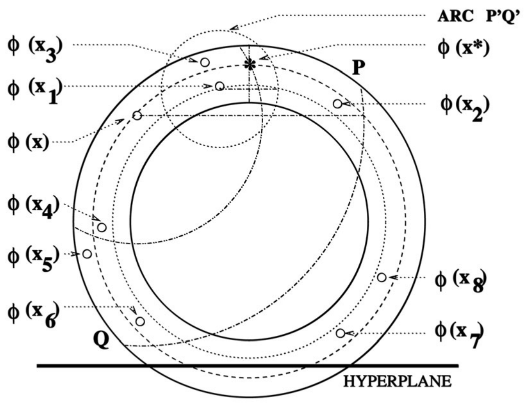

Fig. 9. Arrangement of instances.

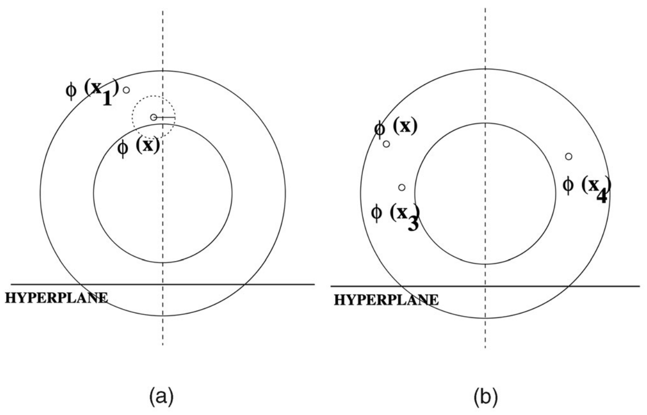

Fig. 10. Errors in determination of farthest instances from hyperplane.

the angular distance between two instances, not their physical order on the ring. Fortunately, pruning out $\phi(\mathbf{x}_5)$ and $\phi(\mathbf{x}_4)$ from the list is simple—we need only remove instances that are closer to the hyperplane than $\phi(\mathbf{x})$. In this case, $\phi(\mathbf{x}_5)$ and $\phi(\mathbf{x}_4)$ are closer to the hyperplane than $\phi(\mathbf{x})$. After removing them from the resorted list, we harvest $\phi(\mathbf{x}_1)$ as the next instance for evaluation. Note that, although $\phi(\mathbf{x}_1)$ is chosen in this cycle, the farthest instance from the hyperplane in the example is actually $\phi(\mathbf{x}_3)$. Next, we use $\phi(\mathbf{x}_1)$ as the anchor instance for the next pruning iteration.

In the second pruning iteration, arc $\mathrm{P'Q'}$ (obtained using the ring associated with $\phi(\mathbf{x}_1)$) is the region that would be examined, anchored by $\phi(\mathbf{x}_1)$. In this step, we use the resorted list of $\phi(\mathbf{x}_1)$ as well as that of its predecessor, $\phi(\mathbf{x})$, to choose the next anchor instance agreed upon by both anchors. We pick the first instance that is common in the resorted lists of all the anchors. In the example, $\phi(\mathbf{x}_1)$ and $\phi(\mathbf{x})$ agree upon selecting $\phi(\mathbf{x}_3)$ as the next "better" instance. The algorithm converges at this point, since we do not have any more instances to examine. At the convergence point, we have obtained three anchor instances: $\phi(\mathbf{x})$, $\phi(\mathbf{x}_1)$, and $\phi(\mathbf{x}_3)$.

We make the following important observations on KDX's intra-ring pruning algorithm:

- At the end of the first iteration, we have indeed found the closest instance to $\phi(\mathbf{x}^*)$ associated with $\phi(\mathbf{x})$. Why do we look for the next anchor instance? Carefully examining Fig. 9, we can see that instance $\phi(\mathbf{x}_3)$, though farther than $\phi(\mathbf{x}_1)$ from $\phi(\mathbf{x}^*)$, is actually farther from the hyperplane than $\phi(\mathbf{x}_1)$. When the dimension of the hypersphere is high and the ring has finite width, we can find instances farther from the hyperplane in many dimensions on the ring's surface.

Consider the ring shown in Fig. 10a. Suppose the next instance chosen is $\phi(\mathbf{x})$, based on the stopping criteria designed by us, it is possible for us to stop at $\phi(\mathbf{x})$. This is because $\phi(\mathbf{x}_1)$ lies outside the arc of interest of $\phi(\mathbf{x})$. The situation can be alleviated somewhat by considering the instances whose angular distances with $\phi(\mathbf{x})$ are less than the value determined by the width of the ring. Our method chooses the closest k neighbors of the best instance found in the ring and updates the current set of top-k instances if necessary. This can induce errors when the top instances in the ring are located as in Fig. 10b. Here, if $\phi(\mathbf{x})$ is found to be the farthest instance in the ring, the choice of top-k closest instances of $\phi(\mathbf{x})$ would prefer $\phi(\mathbf{x}_3)$ over $\phi(\mathbf{x}_4)$. However, in practice, we see that the deviation from the best possible distance values is relatively small. This means that, although the top-k instances selected by KDX may not be exactly the same as the true set of k farthest instances, their distances from the hyperplane are very close to those of the farthest instances.

4.2.6 Finding Starting Instance in Adjacent Ring

Having converged on a suitable instance (the approximate farthest instance) in a ring, we next use the inter-ring index to give us a good starting instance for the next ring. The inter-ring index for an instance contains the closest instance from the adjacent ring(s). Once we obtain the anchor instance, $\phi(\mathbf{x})$, for the new ring, we repeat the intra-ring pruning algorithm in Section 4.2.3. The algorithm terminates when the top-k list is not improved after inspecting multiple rings. The algorithm can also terminate when the wall-clock time allowed to run the top-k query expires.

4.3 KDX-Changing Kernel Parameters

In this section, we discuss methods that allow us to perform indexing using the existing indexing structure when the kernel parameters can change. The form of the kernel function is assumed to remain the same. That is, if we had built the index using the Gaussian kernel, we would continue using the Gaussian kernel, but the parameter σ to the kernel would be allowed to change.

Suppose we wish to look at the ordering of the angles made by instances with a fixed instance say $\mathbf{x}_f$. We are interested in the values taken on by the function $K(\mathbf{x}_i, \mathbf{x}_f)$, where $\mathbf{x}_i$ is any instance in the data set. Consider the Gaussian kernel. The values of interest are given by $K(\mathbf{x}_i, \mathbf{x}_f) = exp(-\frac{\|\mathbf{x}_i - \mathbf{x}_f\|^2}{2\sigma^2})$. Since the exponential function is monotonic in nature, the ordering of instances based on their angular separation from $\mathbf{x}_f$ does not change with a change in parameter σ. The same follows for the Laplacian kernel. The polynomial kernel which has the form $(1 + \mathbf{x}_i \cdot \mathbf{x}_f)^p$ is also monotonic in nature if $p \geq 1$ and $\mathbf{x}_i \cdot \mathbf{x}_f \geq 0, \forall \mathbf{x}_i$.

```
      Input:   Support vectors $\mathbf{z}_i$
               Dataset instances $\mathbf{x}_i$
1:             Intra-ring index $Arr$
               Inter-ring index $inter\_ring$
      Output:  Top-$k$ set $top\_k$
2: $counter = 0$
3: $condition$ = False
4: $top\_k = \{\}$
5: $\theta_c$ = Find_$\theta_c(\mathbf{z}_i, \mathbf{x}_c)$
6: $R$ = Find_ring_of_interest($\theta_c, ring$)
7: $\psi$ = Find_$\psi(\theta_c)$
8: $R' = R$
9: $\mathbf{x}$ = random instance in $R$
10: while $counter < n/g$ and $condition$ = False do
11:   Converged = False
12:   $S = \{\}$
13:   while !Converged do
14:     $(d_1, d_2)$ = Find_distances($\mathbf{x}, \mathbf{w}, \psi$)
15:     $(h, v)$ = Find_$h$_$v(d_1, d_2, \mathbf{x}, \mathbf{x}_c, \mathbf{w})$
16:     $(\tau, \xi)$ = Find_$\tau$_$\xi(h, v)$
17:     $index$ = Bin_search( Arr[$R'$][ inverted_index[$\mathbf{x}$]], $\tau$ )
18:     if $ring[R'][index] == \mathbf{x}$ then
19:       Converged = True
20:     else
21:       $S_{\mathbf{x}}$ = Arrangement($\mathbf{x}$,$\tau$,$\xi$,$R'$)
22:       $\mathbf{x}_n$ = $\cap S$ // Intersection chooses unevaluated instance only
23:       $\mathbf{x} = \mathbf{x}_n$
24:     end if
25:   end while
26:   $condition$ = Ring_termination_condition($top\_k, \mathbf{x}$)
27:   $\mathbf{x}$ = $inter\_ring(\mathbf{x})$
28:   $R'$ = Adjacent($R$)
29:   $counter = counter + 1$
30: end while
```

```
Procedure Adjacent($R$)
1: static $direction = 0$
2: static $num_1 = 1, num_2 = 1$
3: if $direction = 0$ and $R + num_1 < n/g$ then
4:   $R' = R + num_1$
5:   $num_1 = num_1 + 1$
6: else if $R - num_2 \geq 0$ then
7:   $R' = R - num_2$
8:   $num_2 = num_2 + 1$
9: end if
10: $direction = 1 - direction$
11: return $R'$
```

```
Procedure Find_$h$_$v(d_1, d_2, \mathbf{x}, \mathbf{x}_c, \mathbf{w})$
Section IV-B.3
1: $r = sin(cos^{-1}(\phi(\mathbf{x}) \cdot \phi(\mathbf{x}_c))$
2: if $d_1 \times d_2 \geq 0$ and $d_1 \geq 0$ then
3:   $temp = d_2 - d_1$
4:   $v = abs(temp - r)$
5: else if $d_1 \geq 0$ then
6:   $temp = d_1 - d_2$
7:   $v = r + temp$
8: else
9:   $temp = d_2 - d_1$
10:   $v = r - temp$
11: end if
12: $h = \sqrt{r^2 - temp^2}$
```

```
Procedure Find_$\tau$_$\xi(h, v)$
Section IV-B.3
1: $s = \sqrt{h^2 + v^2}$
2: $\tau = cos^{-1}(\frac{2-s^2}{2})$
3: $\xi = cos^{-1}(\frac{2-(2h)^2}{2})$
```

Fig. 11. Effect of inserting new instances.

Replacing $\mathbf{x}_f$ by the central instance, we see that the ordering of instances based on their angular separation with the central instance does not change with change in the kernel parameter. Effectively, this means that the grouping of instances into rings, given a particular form of the kernel function, is invariant with change in the kernel parameter. Further, each row of the intra-ring index is essentially the ordering of the instances in the ring based on their angular separation with the instance associated with that row in the ring. Again, these orderings are unaffected by changes in the value of the kernel parameter. The inter-ring index stores the closest instance from the adjacent ring, the monotonic nature of the kernel functions implies that this index is completely unchanged. Thus, the old indexing structure can be used unchanged by computing only the required values when necessary. Since binary search in an array of size g takes O($log\, g$) time, therefore, the extra computations that need to be performed are of the order O($log\, g$) for each binary search operation.

4.4 KDX-Insertion and Deletion

Insertion into the indexing structure requires the identification of the ring to which the new instance belongs and an update of the indexing structure of the ring. Identification of the ring requires O($log(|G|)$) time, $|G|$ being the number of rings. Updating the index structure within the selected ring requires O(g) time, g being the number of instances in the ring. We are interested in an approximate central

```
Procedure Find_θc($\mathbf{z}_i, \mathbf{x}_c$)
Section IV-B.1
1: $\mathbf{z}_i \leftarrow$ Support vector $i$
2: $\mathbf{w} = \frac{\sum_i^{nsv} \alpha_i y_i \phi(\mathbf{z}_i)}{\sqrt{\sum_{i,j}^{nsv} \alpha_i \alpha_j y_i y_j \phi(\mathbf{z}_i) \cdot \phi(\mathbf{z}_j)}}$
3: $\theta_c = cos^{-1}(\mathbf{w} \cdot \phi(\mathbf{x}_c))$

Procedure Find_ring_of_interest($\theta_c, ring$)
Section IV-B.2
1: for $i = 1$ to $num\_rings$ do
2:     $temp\_array[i] = cos^{-1}(K(ring[i][0], \mathbf{x}_c))$
3: end for
4: $R$ = Bin_Search( $temp\_array, \theta_c$ )

Procedure Find_ψ($\theta_c$)
1: if $\theta_c > \pi/2$ then
2:     $\psi = \pi - \theta_c$
3: else
4:     $\psi = \theta_c$
5: end if

Procedure Find_distances($\mathbf{x}, \mathbf{w}, \psi$)
1: $d = \mathbf{w} \cdot \phi(\mathbf{x})$
2: $d_2 = d/sin(\psi)$
3: $p = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}_c)$
4: $d_1 = p/tan(\psi)$

Procedure Arrangement($\mathbf{x}, \tau, \xi, R'$)
1: $temp\_S = \{\}$
2: $index1$ = Bin_search( $Arr[R'][inverted\_index[\mathbf{x}]], \tau$ )
3: $index2$ = Bin_search( $Arr[R'][inverted\_index[\mathbf{x}]], \xi$ )
4: $counter = 0$
5: while $index1 + counter < index2$ or $index1 - counter > 0$ do
6:     if $index1 + counter < index2$ then
7:         $temp\_S = temp\_S \cup Arr[R'][\mathbf{x}][index1 + counter]$
8:     end if
9:     if $index1 - counter > 0$ then
10:        $temp\_S = temp\_S \cup Arr[R'][\mathbf{x}][index1 - counter]$
11:    end if
12:    $counter = counter + 1$
13: end while
14: return $temp\_S$

Procedure Ring_termination_condition($top\_k, \mathbf{x}$)
1: static $flag = 0$
2: $ring\_top\_k = k$ nearest neighbors of $\phi(\mathbf{x})$
3: for $i = 1$ to $k$ do
4:     Merge $ring\_top\_k$ and $top\_k$
5:     if $top\_k$ modified then
6:         $flag = 0$
7:     else
8:         $flag = flag + 1$
9:     end if
10: end for
11: if $flag == num\_unproductive\_rings$ then
12:     return True
13: end if
14: return False
```

Fig. 12. Algorithm for top-k retrieval continued.

instance, which can roughly ensure that the instances are evenly distributed in each ring. The addition of fresh instances does not disturb this situation and, hence, the recomputation of the central instance is not mandatory. However, when the number of instances added is high compared to the existing data set size then the possibility of a skewed distribution of the instances in the rings is higher. If we assume that the current set of instances in the data set is representative of the distribution of instances, the approximate central instance represents a viable choice even after the insertion of new instances into the database.

Visually, we explain the situation in Fig. 13. The first ring shows instances before insertion. The same ring has been shown in the adjacent figure with a large number of instances added asymmetrically. After the new instances have been added into the index, we resume computation using the old central instance. The search for the most suitable instance within the ring proceeds as before. The new ring with inserted instances can essentially be seen as an asymmetric ring.

If the inserted instances are symmetrically distributed across the ring, there is no effect on the search for the best instance in the ring. In the case where the new instances are mostly added in a small region of the ring, the effect is only seen when the instance of interest lies approximately in that region. The effect is that of having a larger number of possible candidates in the initial step which are pruned out as we approach the best instance. The overall effect is that of slowing the rate of convergence by a few steps. It is important to note that this would happen in the case of any ring which has such uneven distribution of instances and cannot be attributed to the decision not to recompute the

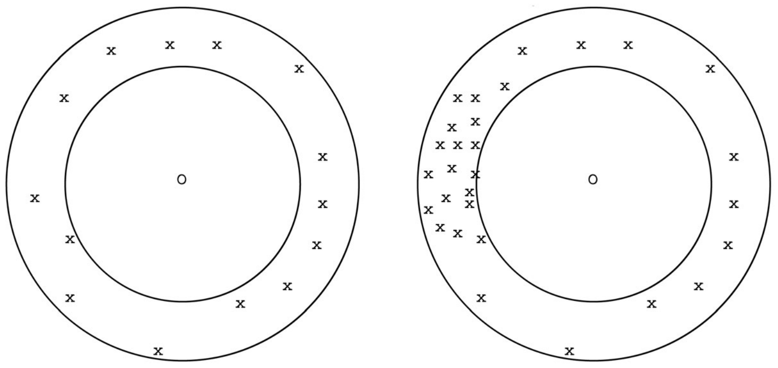

Fig. 13. Effect of inserting new instances.

central instance, since even a recomputation of the central instance does not guarantee that the rings will not have an uneven distribution of instances. When the instance of interest does not lie in the region where most of the new instances lie, the new instances are pruned out at a very early stage thus not affecting the search for the most suitable instance. When the inter-ring index is queried for a good starting point the inserted instances in fact aid by allowing the inter ring index to choose with a finer granularity as compared to before.

The computation of the central instance allows us to divide the instances in the data set into rings so that at least some of the rings have an even distribution of instances. Finding a good starting point speeds up processing in the ring by narrowing down the possible candidates. Essentially, the choice of instance using the inter-ring index would not be as good as in the case of an even distribution, but since we do not rely completely on the inter-ring index for the selection of the best instance in a ring, this effect is at best minimal. Thus, the deletion of instances from a ring does not affect the indexing scheme radically. Deletion of instances involves removing the corresponding row and column from the associated index and takes $O(g)$ time. This cost can be completely avoided if we maintain a record of the deleted items and avoid them when presenting results. Deleted instances which have not been removed are also useful when finding a good starting point in the next ring using the inter-ring index.

5 Experiments

Our experiments were designed to evaluate the effectiveness of KDX using a variety of data sets, both small and large. We wanted to answer the following questions:

- Are the top-k instances chosen by KDX of good quality? (Section 5.1)
- Quantitatively, how good are the results in terms of their distances from the hyperplane? (Section 5.2)
- How effective is KDX in choosing only a subset of the data to arrive at the results? (Section 5.3)
- How does the change in parameters (number of instances per ring and kernel parameter) affect the performance of KDX? (Section 5.4)
- How does the change in center instance affect the performance? (Section 5.5)
- How do data-access and computation strategies affect the cost of retrieval in KDX? (Section 6)

Our experiments were carried out on seven UCI data sets [4], a 21k-image data set, and a 300k-image data set (obtained from Corbis). The seven UCI data sets were selected because of their relatively large sizes; the two selected image data sets have been used in several research prototypes [8]. The details of the data sets are presented in Table 1. In our experiments on top-k retrieval, we obtained results for $k = 10$, 20, and 50 for the Corbis data set, and $k = 20$ for the rest of smaller data sets. The experiments were carried out with the Gaussian kernel.

1. *UCI Data Sets*. We chose seven UCI data sets—namely, Seg, Wine, Ecoli, Yeast, Covtype, vehicle (acoustic), and ijcnn:
    - **Seg**: The segmentation data set was processed as a binary-class data set by choosing its first class as the target class, and all other classes as the

TABLE 1
Data Set Description

Dataset	*# Classes*	*# Training*	*# Testing*
Seg	1	109	103
Wine	3	93	87
Yeast	10	747	737
Ecoli	8	165	171
Covtype	6	57,751	523261
Vehicle(acoustic)	3	7,796	90,730
Ijcnn	1	15,000	91,701
21-k Image	116	4,321	16,983
Corbis	1,173	1,789	312,712

nontarget classes. We then performed a top-k query on the first class.

- **Wine**: The wine recognition data set comes from the chemical analysis of wines grown in the same region of Italy but derived from three different cultivators. Each instance has 13 continuous features associated with it. The data set has 180 instances. We performed three top-k queries on their three classes.
- **Yeast**: The yeast data set is composed of predicted attributes of protein localization sites. The data set contains 1,484 instances with eight predictive attributes and one name attribute. Only the predictive attributes were used for our experiments. This data set has 10 classes, but since the first three classes constitute nearly 77 percent of the data, we used only these three.
- **Ecoli**: This data set also contains data about the localization pattern of proteins. It has 336 instances, each with seven predictive attributes and one name attribute. It has eight classes out of which the first three represent roughly 80 percent of the data and hence were used for our experiments.
- **Covtype**: This is the cover-type data set from the UCI collection of data sets. This data set contains 581,012 instances each with 12 features. It has six classes. The first three were chosen for evaluation.
- **Vehicle (acoustic)**: The feature vectors in this data set represent sound characteristics of different vehicles which are classified into three categories. This data set has 93,044 instances each with 50 features. All three classes in the data set were used for evaluation.
- **Ijcnn**: This data set was used in the IJCNN 2001 competition. This data set has 126,701 instances with 22 features and a single categorization into positive and negative instances.

2. *21-k Image Data Set*. The image data set was collected from the Corel Image CDs. Corel images have been widely used by the computer vision and image-processing communities. This data set contains 21-K representative images from 116 categories. Each image is represented by a vector of 144 features including color, texture, and shape features [8].
3. *Corbis Data Set*. Corbis is a leading visual solutions provider (http://pro.corbis.com/). The Corbis data set consists of more than 300,000 images, each with 144 features. It includes content from museums, photographers, film makers, and cultural institutions. We selected a subset of its more than one thousand concepts.

The number of training and test instances vary slightly with the different classes in the same data set because of differences in the number of positive samples in each class. The samples were randomly picked from both positive and negative classes. In the case of the smaller data sets (Seg, Wine, Yeast, and Ecoli), the percentages of positive and negative samples picked were equal. We chose 50 percent of the entire data set was chosen as training data. For the larger UCI data sets (Covtype, Vehicle (acoustic), and Ijcnn), 10 percent of the data set was chosen as the training data. For the larger data sets (21-k image and the Corbis), the percentage of positive samples picked was higher (50 percent) than the percentage of negative samples chosen. This was done to ensure that the large volume of negative samples does not affect the SVM training algorithm, which is sensitive to imbalances in the sizes of the training and testing data sets. The details of the separation of the data sets are presented in Table 1.

TABLE 2
Qualitative and Quantitative Comparison

Dataset	*% Recall*	*% Discrepancy*	*% Evaluated till recall*	Speedup
Seg	100	0	7.84314	10.14
Wine	93.3	0.27225	22.4806	3.71
Yeast	80.0	0.06603	3.547	21.7
Ecoli	100	0	17.2647	4.1
Covtype	95	0.013	2.13	32.12
Vehicle(acoustic)	78.3	0.12	12.2	6.54
Ijcnn	65	0.25	24.12	3.11
21K	85.0	0.0272883	2.8559	26.29
Corbis	90.0	0.03607813	2.94255	25.43

5.1 Qualitative Evaluation

Given a query, KDX performs a relevance search to return the k farthest instances from the query hyperplane. To measure the quality of the results, we first establish a benchmark by scanning the entire data set to find the top-k instances for each query: This constitutes the "golden" set. The metric we use to measure the query result is *recall*. In other words, we are interested in the percentage of top-k golden results retrieved by KDX.

It is important to note that since we are interested in the approximate set of top-k instances, the exact recall values may in fact be low even if the obtained instances constitute a relevant solution. To address the same, we use a second measure aimed at determining the quantitative nearness of the chosen solution to the best solution possible. This measure is described below.

Results for the qualitative evaluation are presented in the second column of Table 2. The results are averaged over three classes for all the data sets except for Seg. The average recall values for all data sets are above 65 percent. For the Corbis data set, which has a large number of instances each with a large number of features, we have an average recall of 90 percent with less than 4 percent of data evaluated. (We report recall versus fraction of data evaluated in Section 5.3.) The recall values are reasonably high for most of the data sets. There are, however, cases where the discrepancy (described below) values are low but the recall is also low.

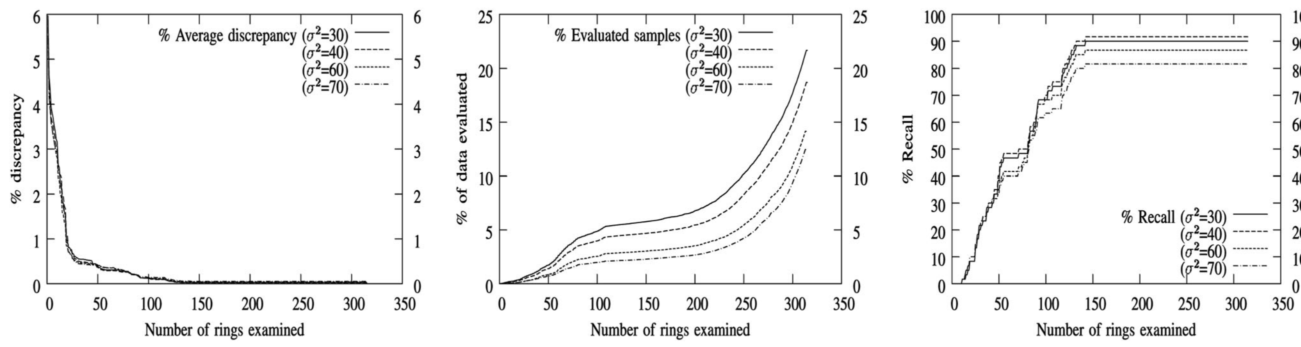

Fig. 14. Corbis data set: variation with change in σ^2 from 30 to 70.

5.2 Evaluation of Discrepancy

This quantitative evaluation involved finding the discrepancy between the average distance to the hyperplane from the top-k instances found by KDX, and the average distance to the hyperplane from the top-k instances in the "golden" set. To obtain a percentage, we divide the average discrepancy by the difference of the distances of the most positive and least positive instances in the data set. The results showing the percentage of average discrepancy for all the data sets are presented in the third column of Table 2. The low values of the percentage of average discrepancy indicate that even if the retrieved instances may not exactly match the golden set of top-k instances, they are comparable in their distances from the hyperplane. None of the data sets has more than 0.3 percent average discrepancy with the values being very low for the large data sets.

5.3 Percentage of Data Evaluated

This evaluation aimed to find the percentage of data evaluated before we obtained the best results using the indexing strategy. The results are reported in the fourth column of Table 2. These values are mostly very low (lower than 10 percent) except in the case of the smaller data sets where, because of the small size of the data set, the percentage of evaluated samples, even with a small number of samples being evaluated, tends to be high. For the large Corbis data set, we find that the results are impressive with less than 4 percent of the data being evaluated to reach 90 percent recall.

We also report the overall speedup achieved over exact evaluation of scores of all instances. In order to compute the speedup, the actual time taken for the computation of the scores of all instances was recorded and compared with the time taken for all the rings for each data set. The computations were carried out on a 1.5GHz processor with 512MB of memory. The speedup figures show a direct correlation with the percentage of instances till recall, with a small overhead because of the evaluation of all rings and index-based computations.

Fig. 15 gives a detailed report of the percentage of average discrepancy, percentage of evaluated samples, and the change in recall as the number of rings increases. In each of the graphs, the x-axis depicts the fraction of the total number of rings processed, and the y-axis depicts the different quantities of interest. The recall (presented in the right-most graph in Fig. 15) reaches a peak early in the evaluation with only a few instances being explicitly evaluated (presented in the middle graph). The discrepancy falls to its lowest level with roughly 4 percent of the data being evaluated (presented in the left-most graph).

5.4 Changes in Parameters

This set of experiments focused on two different parameters. In the first set of experiments, we were interested in evaluating the performance of the indexing strategy when the kernel parameter (in this case, σ of the Gaussian kernel) was changed after the index had been constructed. The second set of experiments evaluated the performance of the indexing strategy when the number of instances per ring was varied.

Fig. 14 shows the results obtained by varying kernel parameter σ^2 between 30 and 70 for the Corbis data set. Here, the x-axis depicts the number of rings examined and the y-axis the quantities of interest (average discrepancy, percentage of data evaluated, and recall). As σ decreases, the angular separation between instances increases, and so does the width of each ring. This affects recall since with wider rings KDX can miss instances as shown in Fig. 10a. However, the extremely low discrepancy values indicate the high quality of the selected instances. Fig. 15 shows the results of changing the number of points in the rings for the Corbis data set from 750 points to 1,500. Though recall generally improves when the number of instances per ring decreases, the percentage of evaluated instances increases. The above results indicate that changes in kernel parameters and number of points in the ring within reasonable limits do not significantly affect KDX's performance.

We also experimented with different k values for the Corbis data set. The results of $k = 10$ and $k = 50$ are reported in Table 3. When k is small, the recall tends to suffer slightly; when k is large, the recall can approximate 100 percent. In both cases, the distance discrepancy remains very small (less than 0.1 percent). Although KDX may occasionally miss a small fraction of the "golden" top-k instances, the quality of the top-k found is very good.

5.5 Effect of Change in Center Instance

Experiments were carried out to explore the effect of a poor choice of center instance. In the experiments, we chose instances at progressively farther distances from the center of the distribution as our center instance. The decay in performance was small. Comparing the performance with

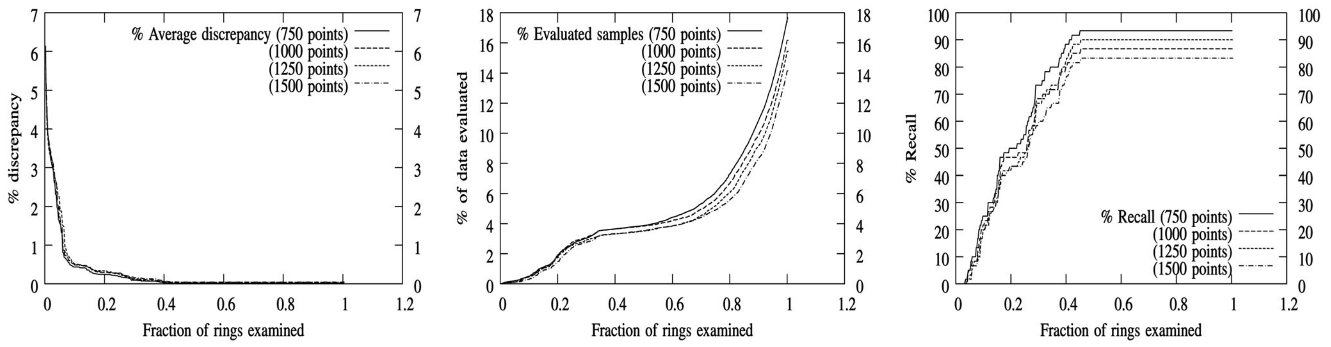

Fig. 15. Corbis data set: variation with change in number of points per ring from 750 to 1,500 ($\sigma^2 = 50$).

two center instances, one chosen closest and the other farthest from the center of the distribution, we found that about 10 extra rings were examined in the latter case to achieve the same level of discrepancy as that achieved in the former case.

5.6 Cost Analysis

In this section, we discuss the costs associated with our indexing strategy in various scenarios. The scenarios we discuss are

- All the rings are examined.
- The best ring has been identified and only a subset of the rings are examined. (This is the scenario we work with in the paper.)
- An alternative strategy is discussed wherein the required arrays are computed as and when required. This strategy is aimed at minimizing the memory consumption.

We discuss each of the scenarios in detail below. However, before that, we would like to outline one important advantage of the index structure in KDX. The partitioning of instances into rings is on the basis of their distance to the central instance only. This allows us to sequentially lay out the partitions on the disc. That is the instances and the associated index structure can be stored on the disc such that all the instances belonging to one partition are placed together. Such a layout enables us to perform sequential access when partitions need to be retrieved from the disc for processing. Sequential access enjoys vastly superior performance when compared to random disc access, often being at least an order of magnitude faster.

TABLE 3
Results with Varying k

Dataset	*Class*	*Recall*	*% Discrepancy*	*% Evaluated till recall*
Corbis	0	0.8	0.05241	3.7729
($k = 10$)	1	1	0	1.82111
	2	0.7	0.119966	2.91755
Corbis	0	0.98	0.000324724	3.83965
($k = 50$)	1	0.96	0.00851683	1.84253
	2	0.9	0.036358	3.06362

5.6.1 Case 1: All Rings Are Examined

Every ring in our indexing scheme maintains an intra-ring index which is essentially a square array of size equal to the number of instances in the ring. Let the instances be equally divided into rings such that each instance contains g instances. The amount of memory consumed by each ring for its intra ring index is $O(g^2)$.

When all the instances as well as the intra-ring indexes can be loaded into memory at the same time at load time, the indexing scheme does not suffer any delays because of disc accesses and the speedup observed depends solely on the number of instances which needed to be completely evaluated. By complete evaluation, we mean that the distance of the instance from the hyperplane was evaluated explicitly using the weighted support vectors. Thus, if there were n instances in the data set and only m needed to be explicitly evaluated by the indexing scheme, then the speedup would be of the order n/m.

Let us consider the case when not all the instances and the associated index structures can be loaded into memory at the same time. Considering only the memory used by the instances without the indexing structure, let it be possible to load n instances out of a total of N instances into memory at the same time. If the number of features associated with each instance is d, then the total memory available is given by $n \times d$. Since each ring consumes $g \times g + g \times d$ space, we can load $\frac{n \times d}{g \times g + g \times d}$ rings into memory at a time. Let the number of rings be c. We have $c = N/g$. The number of disc accesses is given by $\frac{c \times (g \times g + g \times d)}{n \times d}$. This can be simplified to $\frac{N(g \times g + g \times d)}{n \times g \times d} = (1 + \frac{g}{d})\frac{N}{n}$. Thus, the total disc access time is given by $T_D(1 + \frac{g}{d})\frac{N}{n}$, where T_D is the time required for a single disc access. If the fraction of instances that needed to be completely evaluated is F, then the time required to evaluate these is given by $T_C(F \times N \times d \times n_{sv})$, where n_{sv} is the number of support vectors and T_C is the time required for a single floating-point computation. The total time required is given by

$$T_D\Big(1+\frac{g}{d}\Big)\frac{N}{n}+T_C(F\times N\times d\times n_{sv}). \tag{7}$$

The time spent by the sequential algorithm in evaluating all the instances is given by $N\times d\times n_{sv}\times T_C$. The number of disc accesses here is N/n and, therefore, the total time taken is given by

$$T_D\frac{N}{n}+T_C\times N\times d\times n_{sv}. \tag{8}$$

The speedup is given by

$$\frac{T_D\frac{N}{n}+T_C\times N\times d\times n_{sv}}{T_D(1+\frac{g}{d})\frac{N}{n}+T_C(F\times N\times d\times n_{sv})}. \tag{9}$$

It is easy to see that because of the high cost of disc access (T_D) as compared to processing cost (T_C), it is impossible to beat the sequential scan when all the rings are examined. The only situations where it is possible to do so is when all the rings can be loaded into memory at the same time or when the dimensionality of instances is very high compared to g. Such a case is possible when distributed evaluation of queries takes place with the instances in the various rings being spread over multiple computers. This is usually the case for search engines attempting to address user queries quickly.

5.6.2 Case 2: Subset of Rings Examined

This brings us to the strategy used in this paper. Our strategy starts from the most suitable ring for the given query and examines only a subset of all the rings. Since we are starting from the ring which can possibly contain the best instance, examination of a subset of the rings would be enough in most cases to obtain a reasonably good approximate solution. Let the number of rings that are examined be t then the number of disc accesses is given by $\frac{t\times(g^2+g\times d)}{n\times d}$, and the total time taken for this operation is given by $T_D\frac{t\times(g^2+g\times d)}{n\times d}$. The processing time is given by $T_C F t g d n_{sv}$. Hence, the speedup is given by

$$\frac{T_D\frac{N}{n}+T_C\times N\times d\times n_{sv}}{T_D\frac{t\times(g^2+g\times d)}{n\times d}+T_C\times F\times t\times g\times d\times n_{sv}}.$$

We can control the number of disc accesses by controlling the number of rings we examine.

5.6.3 Case 3: Just in Time Computation

The final strategy we outline is the just in time strategy. It is important to note that we only access part of the index that we have created in the preprocessing stage. The rest of the index which is not used consumes memory space without being useful. Since computation cost is much lesser than the cost of disc access we would like to minimize the number of disc accesses by minimizing the size of the index we store.

Here, in addition to the instance vectors we only store the ordering of instances with respect to their angular distance from the central instance and the inter-ring index. In other words, we wish to compute the intra-ring index as and when required. Since we maintain the ordering of instances with respect to the central instance, the instances belonging to any given ring is known.

Having chosen a ring and an instance for evaluation, we need to find its angular separation from the rest of the instances in the ring. This can be accomplished in $O(g\,d)$ time. Sorting the values obtained takes $O(g\log g)$ time. The rest of the steps are essentially the same.

Thus, we have negated the cost of the disc accesses since the number of disc accesses is essentially the same as the number of disc accesses in the case of the sequential access of instances. The computational cost associated is given by $T_C\times F\times N\times(d\times n_{sv}+g\times d+glogg)$.

The computational cost associated with the sequential scan is given by $T_C\times N\times d\times n_{sv}$. Therefore, the speedup here is given by $\frac{d\times n_{sv}}{F\times(d\times n_{sv}+g\times d+glogg)}$.

6 Conclusions

We have presented KDX, a novel indexing strategy for speeding up top-k queries for SVMs. Evaluations on a wide variety of data sets were carried out to confirm the effectiveness of KDX in converging on the approximate set of relevant instances quickly. The improvements were both in terms of the number of distance evaluations performed and the speed of convergence. The quality of the retrieved instances was evaluated both on the basis of the recall and the discrepancy in cumulative distance as compared to the "golden" set of instances farthest from the hyperplane. The indexing structure was also shown to adapt to changing values of the parameters of the kernel function.

As future work, we would like to pursue the goal of further lowering the number of instances to be evaluated. We would also like to develop bounds on the number of instances that KDX evaluates. Our current focus rests on adapting the index structure to obtain the set of uncertain instances in the active learning setting. In such a setting, the query concept is learned in multiple iterations. At each iteration, the user has to be presented with the set of most uncertain instances whose classification can improve the understanding of the concept.

Acknowledgments

This work was supported by US National Science Foundation grants IIS-0133802 and IIS-0219885.

References

[1] C.C. Aggarwal and P.S. Yu, "Outlier Detection for High Dimensional Data," *Proc. ACM SIGMOD Conf.*, 2001.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R^* Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 322-331, 1990.

[3] S. Berchtold, D. Keim, and H.P. Kriegel, "The X-Tree: An Index Structure for High-Dimensional Data," *Proc. 22nd Conf. Very Large Databases*, pp. 28-39, 1996.

[4] C.L. Blake and C.J. Merz, UCI Repository of Machine Learning Databases, 1998.

[5] M. Brown, W. Grundy, D. Lin, N. Christianini, C. Sugnet, M. Jr, and D. Haussler, *Support Vector Machine Classification of Microarray Gene Expression Data*, UCSC-CRL 99-09, Dept. of Computer Science, Univ. of California at Santa Cruz, 1999, http://citeseer.ist.psu.edu/brown99support.html.

[6] J. Chris, C. Burges, and B. Schölkopf, "Improving the Accuracy and Speed of Support Vector Machines," *Advances in Neural Information Processing Systems*, M.C. Mozer, M.I. Jordan, and T. Petsche, eds., vol. 9, p. 375, The MIT Press, 1997.

[7] C.J.C. Burges, "Geometry and Invariance in Kernel Based Methods," *Advances in Kernel Methods,* A.J. Smola, B. Schölkopf, C. Burges, eds., Cambridge, Mass.: MIT Press, 1998.
[8] E. Chang, K. Goh, G. Sychay, and G. Wu, "Content-Based Soft Annotation for Multimodal Image Retrieval Using Bayes Point Machines," *IEEE Trans. Circuits and Systems for Video Technology,* special issue on conceptual and dynamical aspects of multimedia content description, vol. 13, no. 1, pp. 26-38, 2003.
[9] E. Chang and S. Tong, "*Svm_Active*—Support Vector Machine Active Learning for Image Retrieval," *Proc. Ninth ACM Int'l Conf. Multimedia,* pp. 107-118, 2001.
[10] P. Ciaccia, M. Patella, and P. Zezula, "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces," *Proc. 23rd Int'l Conf. Very Large Databases,* pp. 426-435, 1997.
[11] R. Cooley, "Classification of News Stories Using Support Vector Machines," *Proc. 16th Int'l Joint Conf. Artificial Intelligence Text Mining Workshop,* 1999.
[12] C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine Learning,* vol. 20, no. 3, pp. 273-297, 1995.
[13] H. Drucker, D. Wu, and V. Vapnik, "Support Vector Machines for Spam Categorization," *IEEE Trans. Neural Networks,* vol. 10, no. 5, pp. 1048-1054, 1999.
[14] T.S. Furey, N. Duffy, N. Cristianini, D. Bednarski, M. Schummer, and D. Haussler, "Support Vector Machine Classification and Validation of Cancer Tissue Samples Using Microarray Expression Data," *Bioinformatics,* vol. 16, no. 10, pp. 906-914, 2000.
[15] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," *The VLDB J.,* pp. 518-529, 1999.
[16] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene Selection for Cancer Classification Using Support Vector Machines," *Machine Learning,* vol. 46, nos. 1/3, pp. 389-422, Jan. 2002.
[17] T. Joachims, "Text Categorization with Support Vector Machines: Learning with Many Relevant Features," *Proc. 10th European Conf. Machine Learning,* C. Nédellec and C. Rouveirol, eds., pp. 137-142, 1998.
[18] N. Katayama and S. Satoh, "The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," *Proc. ACM SIGMOD Int'l Conf. on Management of Data,* pp. 369-380, 1997.
[19] H. Kim, P. Howland, and H. Park, "Dimension Reduction in Text Classification Using Support Vector Machines," *J. Machine Learning Research,* to appear.
[20] C. Leslie, E. Eskin, and W.S. Noble, "The Spectrum Kernel: A String Kernel for SVM Protein Classification," *Proc. Pacific Symp. Biocomputing,* R.B. Altman, A.K. Dunker, L. Hunter, K. Lauerdale, and T.E. Klein, eds., pp. 564-575, 2002.
[21] C. Li, E. Chang, H. Garcia-Molina, and G. Wilderhold, "Clindex: Approximate Similarity Queries in High-Dimensional Spaces," *IEEE Trans. Knowledge and Data Eng.,* vol. 14, no. 4, July/Aug. 2002.
[22] K.-I. Lin, H.V. Jagadish, and C. Faloutsos, "The TV-Tree: An Index Structure for High-Dimensional Data," *VLDB J.: Very Large Data Bases,* vol. 3, no. 4, pp. 517-542, 1994.
[23] E. Osuna, R. Freund, and F. Girosi, "Training Support Vector Machines: An Application to Face Detection," *Proc. 1997 Conf. Computer Vision and Pattern Recognition (CVPR '97),* pp. 130-138, 1997.
[24] N. Panda and E.Y. Chang, "Exploiting Geometry for Support Vector Machine Indexing," *Proc. SIAM Int'l Conf. Data Mining,* 2005.
[25] P. Pavlidis, J. Weston, J. Cai, and W.N. Grundy, "Gene Functional Classification from Heterogeneous Data," *Proc. Fifth Ann. Int'l Conf. Computational Biology,* pp. 249-255, 2001.
[26] B. Scholkopf, C. Burges, and V. Vapnik, "Extracting Support Data for a Given Task," 1995.
[27] B. Scholkopf, R. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt, *Support Vector Method for Novelty Detection,* pp. 582-588. MIT Press, 2000.
[28] S. Tong and D. Koller, "Support Vector Machine Active Learning with Applications to Text Classification," *Proc. 17th Int'l Conf. Machine Learning,* P. Langley, ed., pp. 999-1006, 2000.
[29] V. Vapnik, *The Nature of Statistical Learning Theory.* Springer Verlag, 1995.
[30] R. Weber, H.-J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," *Proc. 24th Int'l Conf. Very Large Data Bases,* pp. 194-205, 24-27, 1998.

Navneet Panda received the BSc and MSc degrees in mathematics and computing from the Indian Institute of Technology, Kharagpur, in 2000 and 2002, respectively. He has been a PhD student in the Computer Science Department, University of California Santa Barbara since September 2002. His research interests are in the integration of support vector machines and databases for the development of next generation search algorithms and indexing techniques.

Edward Y. Chang received the MS degree in computer science and the PhD degree in electrical engineering from Stanford University in 1994 and 1999, respectively. Since 2003, he has been an associate professor of electrical and computer engineering at the University of California, Santa Barbara. His recent research activities are in the areas of machine learning, data mining, high-dimensional data indexing, and their applications to image databases and video surveillance. Recent research contributions of his group include methods for learning image/video query concepts via active learning with kernel methods, formulating distance functions via dynamic associations and kernel alignment, managing and fusing distributed video-sensor data, and categorizing and indexing high-dimensional image/video information. Professor Chang has served on several ACM, IEEE, and SIAM conference program committees. He cofounded the annual ACM Video Sensor Network Workshop and has cochaired it since 2003. In 2006, he will cochair three international conferences: Multimedia Modeling (Beijing), SPIE/IS&T Multimedia Information Retrieval (San Jose), and ACM Multimedia (Santa Barbara). He serves as an associate editor for the *IEEE Transactions on Knowledge and Data Engineering* and the *ACM Multimedia Systems Journal*. He is a recipient of the IBM Faculty Partnership Award and the US National Science Foundation Career Award. He is a cofounder of VIMA Technologies, which provides image searching and filtering solutions. He is a senior member of the IEEE.