

# Exploiting Geometry for Support Vector Machine Indexing\*

Navneet Panda<sup>†</sup>

Edward Y. Chang<sup>‡</sup>

## 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 dataset is large, naively scanning the entire dataset 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.,  $\gamma$  and  $\sigma$ ) without performance compromise. Through theoretical analysis, and empirical studies on a wide variety of datasets, we demonstrate KDX to be very effective.

## 1 Introduction

Support Vector Machines (SVMs) [6, 19] 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, 11, 14], and query-concept formulation [17, 18]. 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<sup>1</sup> for a concept). KDX supports top- $k$  match as well as top- $k$  uncertainty queries.

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 hyper-rings (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 a couple of important properties. First, it can effectively support insertion and deletion operations. Second, given a kernel function, the indexer works independent of the settings of the kernel parameters (e.g.,  $\gamma$  and  $\sigma$ ). This parameter-invariant property is especially crucial, since varied query-concepts can best be learned under variable parameter settings. Through empirical studies on a wide variety of datasets, 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.

\*Supported by NSF grants IIS-0133802, and IIS-0219885.

<sup>†</sup>Department of Computer Science, UCSB.

<sup>‡</sup>Department of Electrical and Computer Engineering, UCSB.

<sup>1</sup>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.

## 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  $\Phi$ , 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 [20]. Third, a query represented by SVMs is a hyperplane, not a point.

Indexing has been intensively studied over the past few decades. We present some of the representative work in the field but our discussion is by no means exhaustive and for a detailed discussion please consult [13] or [10]. 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 [16], and the SR-tree [12], 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  $\Phi$  is known, the dimension of the projected space could be too high to use the coordinate-based methods due to the curse of dimensionality [15]. 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 [8] 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 the points on the hyperplane 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 [9] 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 [15] but this approach supports only point-based queries, not hyperplane queries.

We developed KDX to effectively tackle the three challenges specified in the beginning of this section.

## 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  $\{\mathbf{x}_1, \dots, \mathbf{x}_{m+n}\}$  that are vectors in some space  $X \subseteq \mathbb{R}^d$ . Among the  $m+n$  instances,  $m$  of them, denoted as  $\{\mathbf{x}_{l,1}, \dots, \mathbf{x}_{l,m}\}$  are assigned labels  $\{y_1, \dots, y_m\}$ , where  $y_i \in \{-1, 1\}$ . The rest are unlabeled data, denoted as  $\{\mathbf{x}_{u,1}, \dots, \mathbf{x}_{u,n}\}$ . 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. The hyperplane is designed to separate 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  $\alpha_i$ 's 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 non-linear decision boundary in the original space  $X$ .) Once the hyperplane has been learned based on the training data  $\{\mathbf{x}_{l,1} \dots \mathbf{x}_{l,m}\}$ , the class membership of an unlabeled instance  $\mathbf{x}_{u,r}$  can be predicted using the  $\alpha_i$ 's of the training instances and their labels  $\{y_1, \dots, y_m\}$

by

$$(3.1) \quad f(\mathbf{x}_{u,r}) = \sum_{i=1}^m \alpha_i y_i K(\mathbf{x}_{l,i}, \mathbf{x}_{u,r}).$$

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,

$$(3.2) \quad K(\mathbf{x}_1, \mathbf{x}_2) = \langle \phi(\mathbf{x}_1), \phi(\mathbf{x}_2) \rangle$$

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

The tunable parameters,  $\sigma$  for Gaussian,  $\gamma$  for the Laplacian kernel, and  $p$  for Polynomial, define different mappings. In each of the above, the mapping function  $\phi$  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 ( $\sigma, \gamma, \text{or } p$ ) 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 while a value of 0 means they are completely dissimilar. The polynomial kernel, though not necessarily normalized, can easily be normalized by using

$$(3.3) \quad K_n(\mathbf{x}_1, \mathbf{x}_2) = \frac{K(\mathbf{x}_1, \mathbf{x}_2)}{K(\mathbf{x}_1, \mathbf{x}_1)K(\mathbf{x}_2, \mathbf{x}_2)},$$

where  $K_n$  is the normalized kernel function. Here, we have assumed that the features associated with each data instance are positive. If not, appropriate normalization needs to be performed.

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 angle 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 ( $\cos^{-1}(0) = \frac{\pi}{2}$ ).

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.

## 4 KDX

In this section, we present our indexing strategy, KDX, for finding the top- $k$  relevant or the top- $k$  uncertain instances (defined shortly) given a hyperplane. We discuss the construction of the index in Section 4.1, the approach for finding the top- $k$  instances in Section 4.2, insertion and deletion operations in Section 4.3, and handling changes in kernel parameters in Section 4.4.

**DEFINITION 4.1.** *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 support vectors, the top- $k$  relevant instances are the set of instances  $(\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_k) \subset S$  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  $\mathbf{q}_1, \dots, \mathbf{q}_k$  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.

**DEFINITION 4.2.** *Top- $k$  Uncertain Instances.* Given the set of instances  $S = \{\mathbf{x}_r\}$ , and the normal to the hyperplane,  $\mathbf{w}$ , represented in terms of the support vectors, the top- $k$  uncertain instances are the set of instances  $(\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_k) \subset S$  such that  $\sum_{i=1, \mathbf{q}_i \in S}^k |\mathbf{w} \cdot \phi(\mathbf{q}_i)|$  is nearest to zero over all possible choices of  $\mathbf{q}_1, \dots, \mathbf{q}_k$  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  $F$ ,

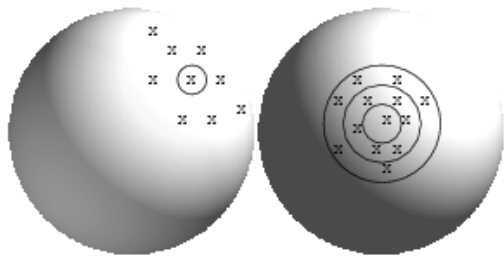


Figure 1: Approximate Central Instance and Rings.

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.

**4.1.1 Finding the central instance** As shown in Figure 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 Equation 3.2).

**LEMMA 4.1.** *The closest approximation of the central instance is the projection of 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 dataset 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  $F$ .

$$\begin{aligned}
 \mathbf{x}_c &= \operatorname{argmin}_{\mathbf{x}_j} \sum_i (\phi(\mathbf{x}_i) - \phi(\mathbf{x}_j))^2 \\
 &= \operatorname{argmin}_{\mathbf{x}_j} \sum_i (\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_i) + \phi(\mathbf{x}_j) \cdot \phi(\mathbf{x}_j) \\
 &\quad - 2\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)) \\
 &= \operatorname{argmin}_{\mathbf{x}_j} \sum_i (2 - 2K(\mathbf{x}_i, \mathbf{x}_j)).
 \end{aligned}$$

Given  $n$  instances in the dataset, each with  $d$  features, finding the central instance in the projected space takes  $O(n^2d)$  time. However, since we are only interested in the approximate central instance, this cost can be easily lowered via a sampling method. 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 being evaluated.

**4.1.2 Separating instances into rings** In this step we compute the angles of the projected instances in  $F$  with the central instance,  $\phi(x_c)$ , using  $K$ . The angles are stored in an array, which is then sorted. 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.

Figure 1 shows the division of instances into different rings. 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, and so on. 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  $i^{\text{th}}$  row of the matrix contains the angles between the  $i^{\text{th}}$  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.

**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 dataset.

## 4.2 KDX-top- $k$

In this section, we describe how KDX finds top- $k$  instances relevant to a query (Definition 4.1) by just examining a fraction of the dataset. Details of the number of instances evaluated are presented in Section 5.

Let us revisit Definition 4.1 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 dataset from the hyperplane, and then selecting the  $k$  instances with greatest distances. This linear-scan approach is clearly costly when the dataset 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  $\theta_c$ , 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- $k$  result for achieving significant speedup. The formal algorithm is presented in Figure 8.

#### 4.2.1 Computing $\theta_c$

Parameter  $\theta_c$  is important for KDX to identify the ring containing the farthest coordinate from the hyperplane. To compute  $\theta_c$ , 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  $\mathbf{w}$  [19]. 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. Next, we compute the angular distance  $\theta_c$  of the central instance  $\phi(\mathbf{x}_c)$  from the normal to the hyperplane.

Given training instances  $\mathbf{x}_{l,1} \dots \mathbf{x}_{l,m}$  and their labels  $y_1 \dots y_m$ , SVMs solve for weights  $\alpha_i$  for  $\mathbf{x}_{l,i}$ . The

normal of the hyperplane<sup>2</sup> can be written as

$$(4.4) \quad \mathbf{w} = \frac{\sum_i^m \alpha_i y_i \phi(\mathbf{x}_{l,i})}{\sqrt{\sum_{i,j}^m \alpha_i \alpha_j y_i y_j \phi(\mathbf{x}_{l,i}) \cdot \phi(\mathbf{x}_{l,j})}}.$$

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

#### 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 4.2. *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  $\theta_c$  obtained in the previous section. We use Figure 2 to illustrate. The figure shows that  $\phi(\mathbf{x}^\diamond)$  is at the intersection of the hypersphere and the normal to the hyperplane with  $\theta_c$  angular separation from  $\phi(\mathbf{x}_c)$ . Given  $\mathbf{x}_c$  and the normal of the hyperplane, we can compute  $\theta_c$  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

<sup>2</sup>Training instances with zero weights are not support vectors and do not affect the computation of the normal.

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 4.2.4 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 Figure 3 depicts the hyperdisc of the current ring. Please note that the hyperdisc can be inclined at an angle to the hyperplane as shown in Figure 4. Back to Figure 3. 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. Figure 3 shows that we need to determine  $h$  and  $v$  in order to use the Pythagoras 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 Figure 4. We discuss in detail how we derive  $s$  in the online version of the paper at <http://www.cs.ucsb.edu/~panda/sdm-complete.pdf>. To focus our presentation on the pruning algorithm, we assume that we have had  $s$  computed.

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 Figure 5. 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 Figure 6 to explain the pruning process. Starting at  $\phi(\mathbf{x})$ , we

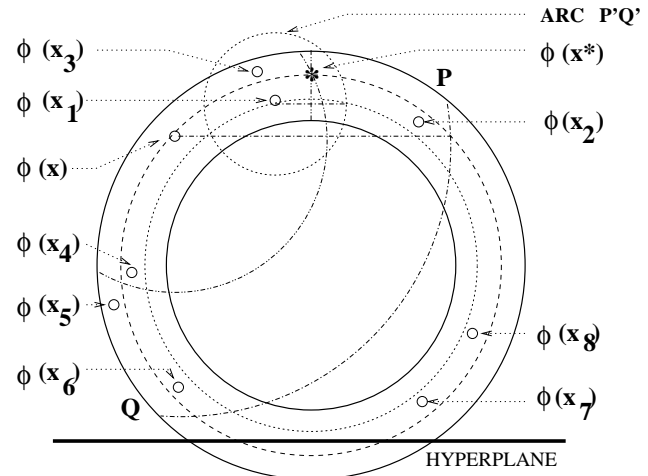


Figure 6: Arrangement of instances

seek to find an instance as close to  $\phi(\mathbf{x}^*)$  as possible. The intra-ring index (Section 4.1.3) of  $\phi(\mathbf{x})$  contains an ordered list of instances based on their distances from  $\phi(\mathbf{x})$ . Let  $\tau$  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  $\tau$  from  $\phi(\mathbf{x})$ . For the example in Figure 6 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 re-sort 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  $\tau$ ). 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 re-sorted list can be obtained by looking up the adjacent instances of the closest instance in the intra-ring index. In our example, this re-sorted 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 re-sorted list. The reason is that we know only 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

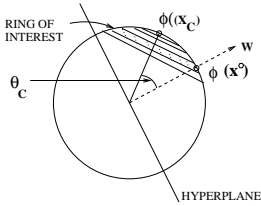


Figure 2: Start ring

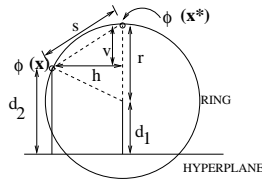


Figure 3: Finding  $s$

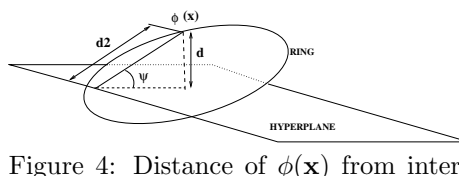


Figure 4: Distance of  $\phi(\mathbf{x})$  from intersection of hyperplane and disc

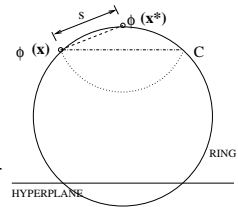


Figure 5: Stopping condition

them from the re-sorted 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 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 re-sorted 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 re-sorted 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 Figure 6, 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.
- In the case of a circle (2D ring with zero width) we can argue about the optimality of the instance chosen by looking at the re-sorted list of the current anchor alone, but the ring in our case is in very high dimensional space and of non-zero width. Therefore, we use information available from any available re-sorted lists of prior anchors in the same ring to validate the choice of the next instance.

Consider the ring shown in Figure 7(a). Suppose the next instance chosen was  $\phi(\mathbf{x})$ , 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 dis-

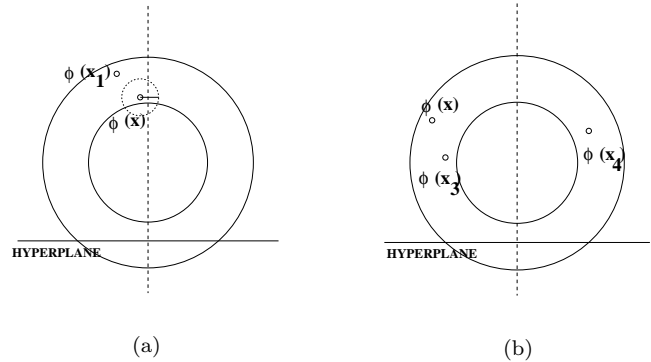


Figure 7: Errors

tances 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 Figure 7(b). 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.

#### 4.2.4 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-insertion and deletion

Insertion into the indexing structure requires the iden-

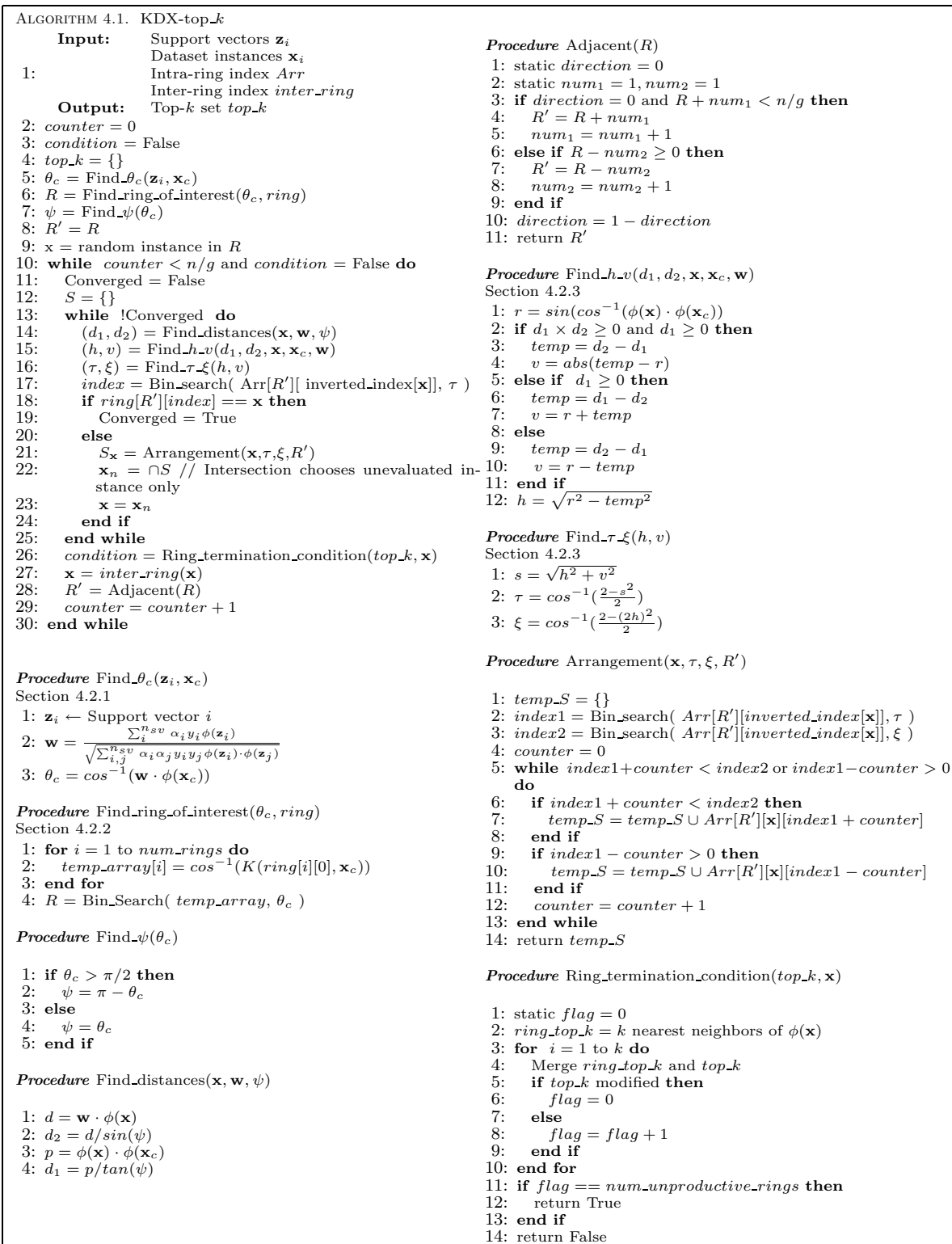


Figure 8: Algorithm for top- $k$  retrieval

tification 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 struc-

ture within the selected ring requires  $O(g)$  time,  $g$  being the number of instances in the ring. Insertion of instances does change the central instance. We are interested in an approximate central instance, which can

roughly ensure that the instances are evenly distributed in each ring. Addition of fresh instances does not disturb this situation, and hence the re-computation of the central instance is not mandatory. However, when the number of instances added is high compared to the existing dataset size, the possibility of a skewed distribution of the instances in the rings is higher. In such a case a re-computation of the central instance and the index would be beneficial. If we assume that the current set of instances in the dataset 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. We discuss the details in the online version of the paper at [http://www.cs.ucsb.edu/~panda/sdm\\_complete.pdf](http://www.cs.ucsb.edu/~panda/sdm_complete.pdf).

#### 4.4 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  $\sigma$  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 dataset. 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  $\sigma$ . 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  $\mathbf{x}_i \cdot \mathbf{x}_f \geq 0, \forall \mathbf{x}_i$ .

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 functioning of the indexing approach outlined before locates a given angle in the sorted array of angles using binary search. Now, after changing the kernel parameter, we do not have the values of the angles which were used to construct the array. But, since the ordering

is unchanged, we can compute the values on the fly when we access an instance in the course of the binary search operation.

Finally, we turn our attention to the inter-group index. Since this index stores the closest instance from the adjacent group, 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.

## 5 Experiments

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

- Are the top- $k$  instances chosen by KDX of good quality?
- Quantitatively, how good are the results in terms of their distances from the hyperplane?
- How effective is KDX in choosing only a subset of the data to arrive at the results?
- How does the change in parameters (number of instances per ring and kernel parameter) affect the performance of KDX?

Our experiments were carried out on four UCI datasets [4], a 21k-image dataset, and a 300k-image dataset (obtained from Corbis). The four UCI datasets were selected because of their relatively large sizes; the two selected image-datasets have been used in several research prototypes [7]. The details of the datasets are presented in Table 1. In our experiments on top- $k$  retrieval we obtained results for  $k = 10, 20$  and  $50$  for the Corbis dataset, and  $k = 20$  for the rest of smaller datasets. The experiments were carried out with the Gaussian kernel.

**UCI Datasets** We chose four UCI datasets—namely, Seg, Wine, Ecoli and Yeast.

**Seg:** The segmentation dataset was processed as a binary-class dataset by choosing its first class as the target class, and all other classes as the non-target classes. We then performed a top- $k$  query on the first class.

**Wine:** The wine recognition dataset 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 dataset has 180 instances. We performed three top- $k$  queries on their three classes.

<i>Dataset</i>	<i># Classes</i>	<i># Training</i>	<i># Testing</i>
Seg	1	109	103
Wine	3	93	87
Yeast	10	747	737
Ecoli	8	165	171
21-k Image	116	4,321	16,983
Corbis	1,173	1,789	312,712

Table 1: Dataset description

**Yeast:** The yeast dataset is composed of predicted attributes of protein localization sites. The dataset contains 1,484 instances with eight predictive attributes and one name attribute. Only the predictive attributes were used for our experiments. This dataset has ten classes, but since the first three classes constitute nearly 77% of the data, we used only these three. **Ecoli:** This dataset 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% of the data and hence were used for our experiments.

**21-k Image dataset** The image dataset was collected from the Corel Image CDs. Corel images have been widely used by the computer vision and image-processing communities. This dataset contains 21-K representative images from 116 categories. Each image is represented by a vector of 144 features including color, texture and shape features [7].

**Corbis dataset** Corbis is a leading visual solutions provider (<http://pro.corbis.com/>). The Corbis dataset consists of over 300,000 images, each with 144 features. It includes content from museums, photographers, filmmakers, 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 dataset 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 datasets (Seg, Wine, Yeast and Ecoli), the percentages of positive and negative samples picked were equal. We chose 50% of the entire dataset was chosen as training data. For the larger datasets (21-k image and the Corbis) the percentage of positive samples picked was higher (50%) 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 datasets. The details of the separation of the datasets are presented in Table 1.

## 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 dataset 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. Results for the qualitative evaluation are presented in the second column of Table 2. The results are averaged over three classes for all the datasets except for Seg. The average recall values for all datasets are above 80%. For the Corbis dataset, which has the largest number of instances, we have an average recall of 90% with less than 4% of data evaluated. (We report recall vs. fraction of data evaluated in Section 5.3.) The recall values are reasonably high for all the datasets.

## 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 dataset. The results showing the percentage of average discrepancy for all the datasets 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 datasets has more than 0.3% average discrepancy with the values being very low for the large datasets.

## 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. In other words, we were interested in finding approximately how quickly KDX converged on its set of best results. The results are reported in the fourth column of Table 2. These values are mostly very low (lower than 10%) except in the case of the smaller datasets where, because of the small size of the dataset, the percentage of evaluated samples, even with a small number of samples being evaluated, tends to be high. For the large datasets, we find that the results are impressive with less than 4% of the data being evaluated to reach 90% recall.

Figure 10 gives a detailed report of the percentage of average discrepancy, percentage of evaluated samples,

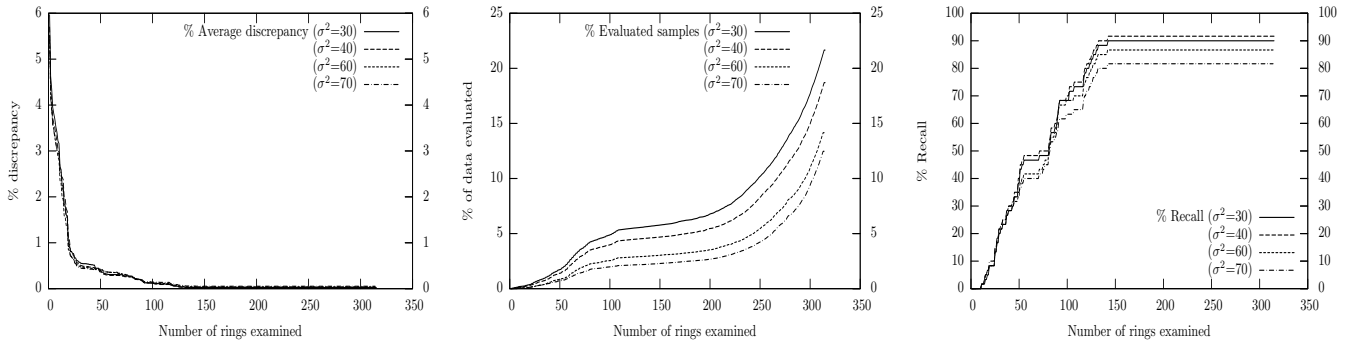


Figure 9: Corbis dataset: variation with change in  $\sigma^2$  from 30 to 70

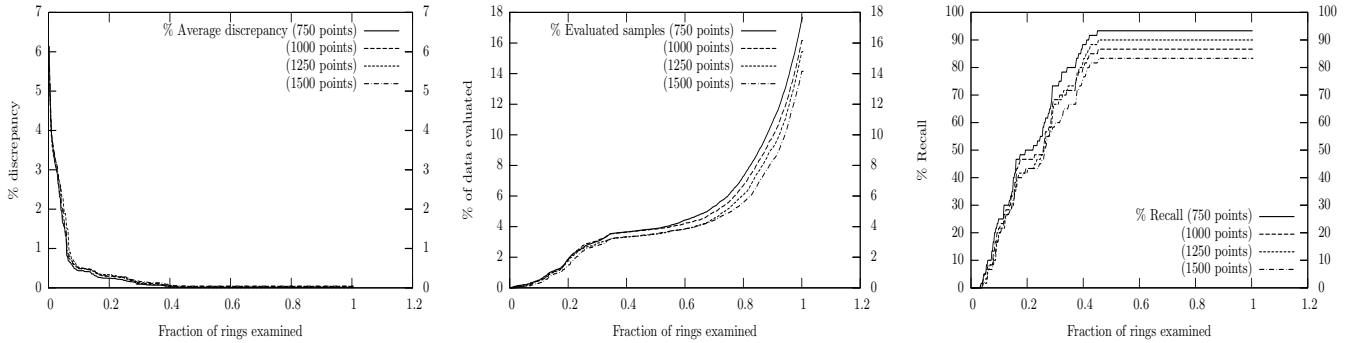


Figure 10: Corbis dataset: variation with change in number of points per ring from 750 to 1500 ( $\sigma^2 = 50$ )

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 Figure 10) 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% 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  $\sigma$  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.

Figure 9 shows the results obtained by varying kernel parameter  $\sigma^2$  between 30 and 70 for the Corbis dataset. 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  $\sigma$  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 Figure 7(a). However, the extremely low discrepancy values indicate the high quality of the selected instances. Figure 10 shows the results of changing the number of points in the rings for the Corbis dataset 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 dataset. 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%. In both cases, the distance discrepancy remains very small (less than 0.1%). Although KDX may occasionally miss a small fraction of the “golden” top- $k$  instances, the quality of the top- $k$  found is very good.

## 6 Conclusions

We have presented KDX, a novel indexing strategy for speeding up top- $k$  queries for SVMs. Evaluations on a wide variety of datasets were carried out to confirm the effectiveness of KDX in converging on relevant instances quickly.

As future work we would like to pursue the goal

Dataset	% Recall	% Discrepancy	% Evaluated till recall
Seg	100	0	7.84314
Wine	93.3	0.27225	22.4806
Yeast	80.0	0.06603	3.547
Ecoli	100	0	17.2647
21K	85.0	0.0272883	2.8559
Corbis	90.0	0.03607813	2.94255

Table 2: Qualitative and quantitative comparison

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

Table 3: Results with varying  $k$

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. Another objective would be to lower the size of the index structure used by KDX. Currently, the index structure takes up  $O(n g)$  space ( $g$  being the number of instances in each ring). Although the dataset itself takes up  $O(n d)$  space ( $d$  being the dimensionality of each feature vector), the size of the index structure can quickly become very large. We would like to explore avenues restricting the size of the index.

## References

- [1] Charu C. Aggarwal and Philip S. Yu. Outlier detection for high dimensional data. In *SIGMOD Conference*, 2001.
- [2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The  $R^*$  tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1990.
- [3] S. BERTHOLD, D. KEIM, and H.P. KRIEGL. The X-tree: An index structure for high-dimensional data. In *22nd Conference on Very Large Databases, Bombay, India*, pages 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. 1999.
- [6] Christopher J.C. Burges. Geometry and invariance in kernel based methods. In Alex J. Smola Bernhard Schölkopf, Chris Burges, editor, *Advances in Kernel Methods*. MIT Press Cambridge, MA, 1998.
- [7] E. Chang, K. Goh, G. Sychay, and G. Wu. Content-based soft annotation for multimodal image retrieval using bayes point machines. *IEEE Trans. on Circuits and Systems for Video Technology Special Issue on Conceptual and Dynamical Aspects of Multimedia Content Description*, 13(1):26–38, 2003.
- [8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. *Proc. 23rd Int. Conf. on Very Large Databases*, pages 426–435, 1997.
- [9] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *The VLDB Journal*, pages 518–529, 1999.
- [10] Michael E. Houle and Jun Sakuma. Fast approximate similarity search in extremely high-dimensional data sets. In *ICDE*, 2004.
- [11] Thorsten Joachims. Text categorization with support vector machines: learning with many relevant features. In Claire Nédellec and Céline Rouveirol, editors, *Proceedings of ECML-98, 10th European Conference on Machine Learning*, number 1398, pages 137–142, Chemnitz, DE, 1998. Springer Verlag.
- [12] Norio Katayama and Shin’ichi Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 369–380, 1997.
- [13] D. A. Keim. Tutorial on high-dimensional index structures: Database support for next decades applications. In *Proceedings of the ICDE*, 2000.
- [14] Hyunsoo Kim, Peg Howland, and Haesun Park. Dimension reduction in text classification using support vector machines. *Journal of Machine Learning Research*, to appear.
- [15] Chen Li, Edward Chang, Hector Garcia-Molina, and Gio Wilderhold. Clindex: Approximate similarity queries in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4), July 2002.
- [16] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal: Very Large Data Bases*, 3(4):517–542, 1994.
- [17] Simon Tong and Edward Chang. Support vector machine active learning for image retrieval. *ACM International Conference on Multimedia*, pages 107–118, 2001.
- [18] Simon Tong and Daphne Koller. Support vector machine active learning with applications to text classification. In Pat Langley, editor, *Proceedings of ICML-00, 17th International Conference on Machine Learning*, pages 999–1006, Stanford, US, 2000. Morgan Kaufmann Publishers, San Francisco, US.
- [19] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer Verlag, 1995.
- [20] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 194–205, 24–27 1998.