Exact Indexing for Support Vector Machines

Hwanjo Yu*, Ilhwan Ko, Youngdae Kim, Seungwon Hwang
Department of Computer Science and Engineering
POSTECH
Pohang, South Korea
{hwanjoyu,koglep,prayer,swhwang}@postech.ac.kr

Wook-Shin Han
Department of Computer Science and Engineering
Kyungpook National University
Daegu, South Korea
wshan@knu.ac.kr

ABSTRACT

SVM (Support Vector Machine) is a well-established machine learning methodology popularly used for classification, regression, and ranking. Recently SVM has been actively researched for rank learning and applied to various applications including search engines or relevance feedback systems. A query in such systems is the ranking function F learned by SVM. Once learning a function F or formulating the query, processing the query to find top-k results requires evaluating the entire database by F.

So far, there exists no *exact* indexing solution for SVM functions. Existing top-k query processing algorithms are not applicable to the machine-learned ranking functions, as they often make restrictive assumptions on the query, such as linearity or monotonicity of functions. Existing metric-based or reference-based indexing methods are also not applicable, because data points are invisible in the kernel space (SVM feature space) on which the index must be built. Existing kernel indexing methods return approximate results or fix kernel parameters. This paper proposes an exact indexing solution for SVM functions with varying kernel parameters.

We first propose key geometric properties of the kernel space – *ranking instability* and *ordering stability* – which is crucial for building indices in the kernel space. Based on them, we develop an index structure *iKernel* and processing algorithms. We then present clustering techniques in the kernel space to enhance the pruning effectiveness of the index. According to our experiments, *iKernel* is highly effective overall producing 1∼5% of evaluation ratio on large data sets. According to our best knowledge, *iKernel* is the first indexing solution that finds *exact* top-k results of SVM functions without a full scan of data set.

Categories and Subject Descriptors

H.m [**Information Systems**]: Miscellaneous

General Terms

Algorithms, Performance

*This work was supported by the Brain Korea 21 Project in 2010 and Mid-career Researcher Program through NRF grant funded by the MEST (No. KRF-2009-0080667).

SIGMOD'11, June 12–16, 2011, Athens, Greece.

Keywords

Support Vector Machines, Indexing

1. INTRODUCTION

Motivation: SVM (Support Vector Machine) is a well-established machine learning methodology popularly used for classification, regression, and ranking [17, 3]. Recently SVM has been actively researched for rank learning and applied to search engines or relevance feedback systems [4, 16, 14, 19, 21, 22, 20, 23]. For example, in content-based image retrieval (CBIR) systems, the user provides feedback of whether the resulting images are relevant or not, from which SVM learns a ranking function F, and the function is evaluated through the database to return top-k relevant images. A query in this case is the ranking function F learned by SVM.

Researchers in SVM community have focused on improving the accuracy or learning efficiency of SVM and developed numerous algorithms for it, but they pay relatively less attention to processing the SVM function for efficiently finding top-k results. Processing or testing an SVM function on a large set of data is crucial in many applications and takes a non-trivial amount of time. For example, in the above retrieval systems, learning a function F (or formulating the query) is done instantly since the training data is typically small, while processing the query to find top-k results requires evaluating the entire database by F.

So far, there exists no *exact* indexing and processing methods for SVM functions. Existing top-k processing algorithms are not applicable to the machine-learned ranking function, as they often make restrictive assumptions on the query, such as linearity or monotonicity of functions [8, 9, 2, 10]. Existing metric-based or reference-based indexing methods [7, 1, 11] are also not applicable, because data points are invisible and the distance function changes with queries in the kernel space (SVM feature space) where the index must be built. Existing kernel indexing methods assumed to have a fixed distance function (or a fixed kernel parameter) [16] or return approximate results [15]. (We will detail related work in Section 2.) This paper proposes an exact indexing and processing methods for SVM, which quickly find top-k results without scanning the entire database.

Background: Ranking functions learned by SVMs are *kernel ranking functions*, which are found to be of *simple* structure but are highly *expressive* in representing the user's hidden ranking. More specifically, a ranking function $F(\vec{z})$ returns a ranking score of instance (or vector) $\vec{z}$ such that higher ranking score indicates higher preference or relevance. A kernel ranking function F is structured as

$$F(\vec{z}) = \sum \alpha_i \mathcal{K}(\vec{x}_i, \vec{z}) \quad (1)$$

where $\vec{x}$ and α are respectively some instances (support vectors in SVMs) and their coefficients, and the kernel $\mathcal{K}$ returns a similarity score $(0, 1]$ between $\vec{x}_i$ and $\vec{z}$. For example, $\mathcal{K}$ for the RBF kernel, *i.e.*, the most populary used nonlinear kernel, is $\mathcal{K} = exp(-s||x_i - z||^2)$. (Backgrounds on kernel ranking function are detailed in Section 3 and 4.)

Our specific goal is to quickly find top-k instances $\vec{z}$ of highest $F(\vec{z})$ among all instances. Note that, evaluating one nonlinear kernel ranking function $F(\vec{z})$ typically takes a non-negligible time, as it involves scanning through the support vectors. Thus, *the processing time is usually dominated by the evaluation time*, as we will also demonstrate with experimental results in Section 6.

Challenges: Toward the goal, we first observe common properties of the nonlinear kernel ranking functions. Specifically, for all nonlinear kernel ranking functions, a kernel $\mathcal{K}$ can be represented as a dot product of two vectors in a "feature space", and a complex nonlinear ranking function in data space is represented as a linear function in the feature space. Prior literature [13] noted that, data instances in the feature space are scattered on the *surface* of a hypersphere, and the ranking function is represented as the normal vector to a hyperplane crossing the center of the hypersphere. Top results are the instances farthest from the hyperplane or nearest to the normal vector. Figure 1 illustrates an example of data instances (points) and a ranking function (the normal to a hyperplane) in a three-dimensional feature space. In this surface space, the top-k processing problem is translated into finding k nearest neighbors to a query point q (*i.e.*, the normal vector). (Details are discussed in Section 4.)

In this feature space, however, data instances are only defined with respect to pairwise distances, *i.e.*, angle distances between instances, and their absolute feature values are "undefined" and thus cannot be visualized. (Figure 1 shows "hypothetical visualization" for illustration purposes only.) Thus, the well-known nearest neighbor algorithms built on indices over absolute values, such as R-tree, X-tree, TV-tree, SR-trees, are not applicable. Instead, we need to identify nearest neighbors in a space where distances are only defined relatively for pairs.

Reference-based algorithms [7, 1, 11] have been studied for the metric space where the pairwise distance function is fixed. However, our target problem poses another key challenge. *Since kernel parameters changes over varying queries, the pairwise distance function changes too!* While more recent work tackled this challenge [13, 14], existing algorithms find *approximate* top-k results. To the best of our knowledge, no method exists which returns *exact* top-k results of a query of kernel ranking functions.

Contributions: To build an exact index structure in the kernel space where the feature values are hidden and the distance function changes with queries, we first propose key geometric properties of the kernel space – *ranking instability* and *ordering stability* – which is crucial for building indices in the kernel space. The properties prove that although the distance function changes with queries in the kernel space, the ordering of distances does not change.

Based on the properties, we propose a novel index structure *iKernel*, which is a set of "reference-based rings". The ring structures are necessary for indexing in the kernel space, as they keep the orderings of instances rather than the actual distances. In other words, *iKernel* builds an index using only the *ordering* information, which is *invariant* to the kernel parameters, thus unchanging over queries.

We then propose an efficient processing algorithm using the index to identify exact top-k results for a query of kernel ranking function. We formally discuss both the correctness and optimality of our proposed algorithm. Finally, we propose a density-based clustering technique in the kernel space in order to enhance the pruning effectiveness of the index. According to our experiments, *iKernel* is highly effective overall producing 1∼3% of evaluation ratio on large data sets, and its index size and construction time is substantially smaller and faster than the existing approximate solution. Also the maintenance for inserting and deleting new instances is inexpensive. According to our best knowledge, *iKernel* is the first indexing solution that finds *exact* top-k results of SVM functions without a full scan of data set.

Organization: This paper is organized as follows. We first discuss related work (Section 2). Section 3 explains fundamentals of kernel ranking functions. Section 4 presents the key geometric properties of kernel ranking functions that are crucial in building and proving the correctness of our methods. Section 5 details our proposed methods. Section 6 reports experimental evaluations. Section 7 concludes our study.

2. RELATED WORK

Top-k processing algorithms: Most existing top-k query processing algorithms [8, 9, 2, 10] generally assume that ranking function F is (1) defined over absoluate attribute values and (2) monotonic over values. Building upon these assumptions, existing algorithms achieve the efficiency by exploiting generic similarity index structures, such as B-trees on attributes, to selectively access the high-scoring sub-region. More recent efforts [24, 18] have focused on relaxing the monotonicity assumption to include functions whose scores can be bounded in the given attribute value range. However, these algorithms cannot be applied to our target problem of supporting nonlinear kernel functions violating the standard assumptions.

Indexing for generic similarity ranking: While numerous high-dimensional indices for similarity ranking have been proposed, such as R-tree, X-tree, TV-tree, and SR-tree as surveyed in [12], these structures assume all *absolute values* of ranking attributes of data instances are known, based on which they partition the attribute space into regions for pruning out irrelevant regions or narrowing down to relevant regions. In a clear contrast, as Section 3 discusses, absolute attribute values for nonlinear kernel functions are undefined, which makes it infeasible to apply existing index structures.

However, once transformed into feature space, even though absolute values are still unknown, we can obtain relative distances. While indexing for relative distances (from reference points) has been studied in [7, 1, 11], these indices are also not applicable to our target problem where such distances change over queries.

Indexing for SVM functions: Several indexing methods that are specialized for SVM functions have been proposed [16, 13, 14] with the following restrictions. The index proposed in [16] assumes kernel parameters are fixed, which frequently change over queries in our target system. Another approaches [13, 14] aim at approximate processing, by clustering data instances in the feature space based on relative proximity to the global centroid of data instances. They compute top-k nearest points to the query point based on their relative distances to the global centroid. However, these approaches are approximate and cannot ensure the correctness, because the relative distances do not tell any directional information. For example, given δ = the distance between query point and the centroid, points that are away from the centroid by δ may not be close to the query point if they are away from the centroid in different directions. According to our experiments, their approximation is sometimes very rough especially on large data sets. Moreover, their index size and construction time is substantially larger than ours.

3. KERNEL RANKING FUNCTION

This section presents preliminary background of the kernel ranking function. Suppose there is a set of data $\{\vec{x}\}$ that are vectors in a "data space." The kernel ranking function in Eq.(1) is composed of support vectors $\vec{x}$ and their coefficients α. The kernel $\mathcal{K}$, a kind of similarity function, returns a dot product of two vectors in some "feature space," *i.e.*,

$$\mathcal{K}(\vec{x}_1, \vec{x}_2) = \phi(\vec{x}_1) \cdot \phi(\vec{x}_2) \quad (2)$$

where ϕ is an implicit mapping used for projecting the data space instances $\vec{x}_1$ and $\vec{x}_2$ onto the feature space. Thus, the function of Eq.(1) becomes the following.

$$F(\vec{z}) = \sum \alpha_i \mathcal{K}(\vec{x}_i, \vec{z}) = \sum \alpha_i \phi(\vec{x}_i) \cdot \phi(\vec{z}) = \vec{W} \cdot \phi(\vec{z}) \quad (3)$$

where $\vec{W}(= \sum \alpha_i \phi(\vec{x}_i))$ is the weight vector (or the query point) in the feature space. From another point of view, F is represented by the vector (or point) $\vec{W}$ in the feature space, and the rankings of instances are determined based on their dot products with $\vec{W}$ in the feature space. Note that *F is a linear function in the feature space, but it becomes nonlinear in the original data space.* The mapping function ϕ is not defined explicitly. Instead, we use the *"kernel trick."* which replaces the dot product of two vectors in the feature space with a kernel which is a function of two vectors in the original data space. Computing the kernel $\mathcal{K}$ of two vectors in the data space is simple and efficient. The following illustrates an example of the kernel trick.

EXAMPLE 1 (KERNEL TRICK). *Assume that the data space is two-dimensional, and for two vectors $\vec{a} = [a_1, a_2]$ and $\vec{b} = [b_1, b_2]$, we compute their feature space dot product by the following kernel.*

$$\phi(\vec{a}) \cdot \phi(\vec{b}) = \mathcal{K}(\vec{a}, \vec{b}) = (\vec{a}\vec{b} + 1)^2$$

Then,

$$\mathcal{K}(\vec{a}, \vec{b}) = 1 + a_1^2 b_1^2 + 2a_1 a_2 b_1 b_2 + a_2^2 b_2^2 + 2a_1 b_1 + 2a_2 b_2$$

Thus, the vector $\vec{a}$ projected onto the feature space $\phi(\vec{a})$ is deduced to be

$$\phi(\vec{a}) = [1, a_1^2, \sqrt{2}a_1 a_2, a_2^2, \sqrt{2}a_1, \sqrt{2}a_2]$$

Similarly,

$$\phi(\vec{b}) = [1, b_1^2, \sqrt{2}b_1 b_2, b_2^2, \sqrt{2}b_1, \sqrt{2}b_2]$$

While the original data space is two-dimensional, the feature space becomes six-dimensional in this example. Note that, by computing the kernel with the data space vectors, i.e., *$(\vec{a}\vec{b} + 1)^2$, we do not need to explicitly compute the ϕ or project the vectors onto the feature space.*

In the above example, the feature space $\phi(\cdot)$ can be computed but this incurs expensive cost. However, in practice, computing $\phi(\cdot)$ is not typically possible at all. By using the kernel trick, such an expensive feature space computation is replaced by a simple data space computation. The following are popularly used nonlinear kernels, and the RBF kernel is especially popular because it is proven to have near-infinite expressiveness [3].

RBF kernel:	$\mathcal{K}(\vec{x}_1, \vec{x}_2) = exp(-s\|\|\vec{x}_1 - \vec{x}_2\|\|^2)$
Laplacian kernel:	$\mathcal{K}(\vec{x}_1, \vec{x}_2) = exp(-s\|\vec{x}_1 - \vec{x}_2\|)$
Polynomial kernel:	$\mathcal{K}(\vec{x}_1, \vec{x}_2) = (\vec{x}_1 \vec{x}_2 + 1)^t$

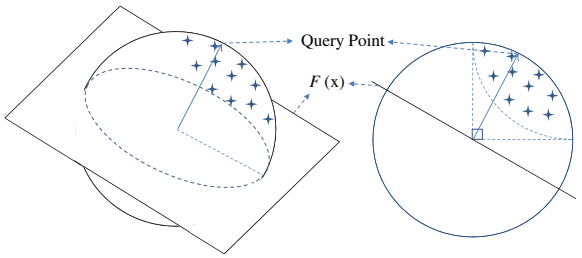


Figure 1: Examples of instances in a three-dimensional feature space

The kernel parameters, $s(> 0)$ for RBF and Laplacian, and t (natural number) for Polynomial, define different mappings. That is, *the similarities (or distances) between data pairs in the feature space, computed by the dot products, change along with the kernel parameters.* The following section will delineate the impacts of the kernel parameters on designing top-k index.

4. PROPERTIES OF KERNEL RANKING FUNCTIONS

This section presents several important properties of kernel ranking functions and proposes lemmas that are crucial in building and proving the correctness of our *iKernel*.

1. The instances projected to the feature space, although they cannot be seen, lie on the surface of a unit hypersphere, and the angle of any two instances is bounded by $\frac{\pi}{2}$ [15]. It is because the cosine similarity of any two instances in the feature space, that is returned by a kernel, is between 0 and 1, *i.e.*, $0 \leq \phi(\vec{x}_1) \cdot \phi(\vec{x}_2) = \mathcal{K}(\vec{x}_1, \vec{x}_2) \leq 1$, and the dot product of an instance with itself in the feature space is equal to 1, *i.e.*, $\mathcal{K}(\vec{x}, \vec{x}) = 1$. For example, Figure 1 shows an example of instances in a three-dimensional feature space. RBF kernel returns one when two vectors are identical while it returns a value close to zero as two vectors become dissimilar. Other nonlinear kernels including Polynomial kernel can be normalized as follows.

$$\mathcal{K}'(\vec{x}_1, \vec{x}_2) = \frac{\mathcal{K}(\vec{x}_1, \vec{x}_2)}{\sqrt{\mathcal{K}(\vec{x}_1, \vec{x}_1)\mathcal{K}(\vec{x}_2, \vec{x}_2)}}$$

   where $\mathcal{K}'$ returns a value between 0 and 1 and is a valid kernel reproducing kernel Hilbert space (RKHS) [6] when the attributes are equal or greater than zero. (If not, proper scaling of the attributes can be performed.)

2. Since ϕ in Eq.(2) is an implicit mapping, we cannot see the vectors projected to the feature space. Instead, *we can only see "parameterized" distances among instances in the feature space.* The distance of two vectors in the feature space is measured by their angle difference, as they lie on the surface of a unit hypersphere. That is, the angular distance D($\vec{x}_1, \vec{x}_2$) of $\vec{x}_1$ and $\vec{x}_2$ is

$$cos^{-1}(\phi(\vec{x}_1) \cdot \phi(\vec{x}_2)) = cos^{-1}(\mathcal{K}(\vec{x}_1, \vec{x}_2)) \quad (4)$$

   This distance measure satisfies the metric properties such as symmetry and triangular inequality. Note that the kernel $\mathcal{K}$ is typically fixed for an application, but the kernel parameter is determined at query time. For example, if we build an index using RBF kernel for some applications, we would continue

using the RBF kernel for those applications but the parameter s would change each time a query or a kernel ranking function F is formulated.

We first show, by Lemma 1 below, that the ranking of the instances given a kernel ranking function α and $\vec{x}_i$ in Eq.(1) does change with the kernel parameter.

LEMMA 1 (RANKING INSTABILITY). *Given support vectors $\vec{x}$, their coefficients α, and either of RBF, Laplacian, or normalized Polynomial kernel $\mathcal{K}$, the ranking of instances $\vec{z}$ according to the kernel ranking function $F = \sum \alpha_i \mathcal{K}(\vec{x}_i, \vec{z})$ is **variant** to the kernel parameter.*

PROOF. As for the RBF kernel, in order for the ranking of instances $\vec{z}$ to be *not* variant to the parameter, for any support vectors $\vec{x}$, their coefficients α, and any tuple of instances $\vec{z}_1$ and $\vec{z}_2$, if $\sum \alpha_i exp(-s_1||\vec{x}_i - \vec{z}_1||^2) > \sum \alpha_i exp(-s_1||\vec{x}_i - \vec{z}_2||^2)$, then an inequality, $\sum \alpha_i exp(-s_2||\vec{x}_i - \vec{z}_1||^2) > \sum \alpha_i exp(-s_2||\vec{x}_i - \vec{z}_2||^2)$, must satisfy for all positive value s_1 and s_2. However, we can easily find a counter example such as, $\alpha_1 = 0.5, \alpha_2 = 1, \vec{x}_1 = 2, \vec{x}_2 = 0, \vec{z}_1 = 1, \vec{z}_2 = 2, s_1 = 1, s_2 = 4$. Intuitively, when s is large in RBF kernel, F increases as $\vec{z}$ gets closer evenly to all $\vec{x}$ rather than one $\vec{x}$. The counter example also works for Laplacian kernel for the similar reason. We can also find a counter example for the normalized Polynomial kernel such as,$\alpha_1 = 0.5, \alpha_2 = 0.5, \vec{x}_1 = [1, 0], \vec{x}_2 = [0, 1], \vec{z}_1 = [1, 0], \vec{z}_2 = [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}], t_1 = 1, t_2 = 5$. Thus, the ranking of instances according to the kernel ranking function with RBF, Laplacian, or normalized Polynomial kernels is variant to the kernel parameter. □

While Lemma 1 states that *the kernel parameters determined at query time actually affect the ranking of instances*, we observe the *invariant* property which plays a key role in designing indices to be built upon such invariant information. More formally, Lemma 2 states that the *ordering of distances* among instances in the feature space does *not* change with the kernel parameter, although the actual distances change. Inspired by Lemma 2, we later design our indices to keep track of *ordering of distances* staying invariant over queries, instead of the actual distances changing over queries.

LEMMA 2 (ORDERING STABILITY). *Given either of RBF, Laplacian, or normalized Polynomial kernel $\mathcal{K}$ and a set of instances $\mathcal{X}$, the ordering of distances between $\vec{x}(\in \mathcal{X})$ and the other instances in $\mathcal{X}$, measured by the angular distance $cos^{-1}(\mathcal{K}(\vec{x}_1, \vec{x}_2))$, is **invariant** to the kernel parameter.*

PROOF. To prove for RBF kernel, for any triplet of vectors $\vec{x}_1$, $\vec{x}_2$, $\vec{x}_3 \in \mathcal{X}$, if

$$cos^{-1}(exp(-s_1||\vec{x}_1 - \vec{x}_2||^2)) > cos^{-1}(exp(-s_1||\vec{x}_1 - \vec{x}_3||^2)),$$

then the inequality

$$cos^{-1}(exp(-s_2||\vec{x}_1 - \vec{x}_2||^2)) > cos^{-1}(exp(-s_2||\vec{x}_1 - \vec{x}_3||^2))$$

always satisfies for any positive value s_1 and s_2, because,

$$\begin{aligned} cos^{-1}(exp(-s_1||\vec{x}_1-\vec{x}_2||^2)) > cos^{-1}(exp(-s_1||\vec{x}_1-\vec{x}_3||^2)) \Rightarrow \\ exp(-s_1||\vec{x}_1 - \vec{x}_2||^2) < exp(-s_1||\vec{x}_1 - \vec{x}_3||^2) \Rightarrow \\ ||\vec{x}_1 - \vec{x}_2||^2 > ||\vec{x}_1 - \vec{x}_3||^2 \Rightarrow \\ exp(-s_2||\vec{x}_1 - \vec{x}_2||^2) < exp(-s_2||\vec{x}_1 - \vec{x}_3||^2) \Rightarrow \\ cos^{-1}(exp(-s_2||\vec{x}_1-\vec{x}_2||^2)) > cos^{-1}(exp(-s_2||\vec{x}_1-\vec{x}_3||^2)). \end{aligned}$$

Similarly we can prove it for the Laplacian kernel and normalized Polynomial kernel. Thus, the ordering of the similarities measured by the RBF, Laplacian, or normalized Polynomial kernel is invariant to the parameter. □

5. IKERNEL

This section first defines the problem and overviews our approach to solve the problem (Section 5.1), and discusses in details (1) how to construct the index (Section 5.2), (2) how to process the top-k query of nonlinear kernel ranking functions (Section 5.3), and (3) how to cluster the data to improve the pruning ratio (Section 5.4). We then discuss the time complexity for indexing and processing, and discuss insertion and deletion operations (Section 5.5).

5.1 Overview of iKernel

As shown in Eq.(3) of Section 3, a query is represented as a vector $\vec{W}$, *i.e.*, the normal to a hyperplane or a point on the surface of a unit hypersphere as discussed in Section 4. Note that, as discussed in Section 3, $\vec{W}$ is not explicitly known since all operations in the feature space are only implicitly performed through the kernel. A query is in fact given as a set of α_i and $\vec{x}_i$, and the kernel parameter, which constitutes the query point $\vec{W}$. Thus, our problem is defined as follows.

DEFINITION 1. *Given a set of instances $\mathcal{X} = \{\vec{x}\}$ and a query point $\vec{W}$ represented as the support vectors, their coefficients, and the kernel parameter, the top-k instances are the k instances $\mathcal{S} \subset \mathcal{X}$ such that, for all $\vec{a} \in \mathcal{S}$ and $\vec{b} \notin \mathcal{S}$ and $\vec{a}, \vec{b} \in \mathcal{X}$, $cos^{-1}(\vec{W} \cdot \phi(\vec{a})) < cos^{-1}(\vec{W} \cdot \phi(\vec{b}))$.*

A key challenge here is that the feature space distances of Eq.(4) between instances change with the kernel parameter, as we elaborated in Section 4. Thus, *we need to build the indexing method that is invariant to the kernel parameter, and also must be effective for pruning a large amount of instances given a kernel ranking function, without storing the actual distances.*

The key idea of *iKernel* is to keep track of the ordering of distances between some reference points and the other instances in the feature space. According to Lemma 2, the ordering of distances does not change with the kernel parameters thus is independent of queries. Specifically, *iKernel* starts constructing rings from each reference points where each ring contains g instances, and the instances within each ring are sorted according to *their orderings* from the reference points. When a query point q is given, the distance between q and the farthest instance at each ring is computed in order to determine whether to prune the ring. Since the actual distance among instances are unknown until the query time, it is necessary to maintain the rings of instances in which the instances are sorted according to the distance ordering.

Determining the reference points must be done with caution since it could significantly affect the pruning ratio. We will propose a density-based clustering algorithm in Section 5.4 that, using only the parameterized distances, determines the reference points (*i.e.*, centroids of density-based clusters) that improve the pruning ratio. The process of *iKernel* can be divided as follows.

- Offline processing: Index construction
  1. Compute density-based clusters and the centroids (locally peak points) in the feature space (Section 5.4).
  2. Build an index using the centroids and clusters (Section 5.2).
- Online processing: Top-k query processing
  1. Given a query of kernel ranking function, use the index to determine top-k results with minimal probing (Section 5.3).

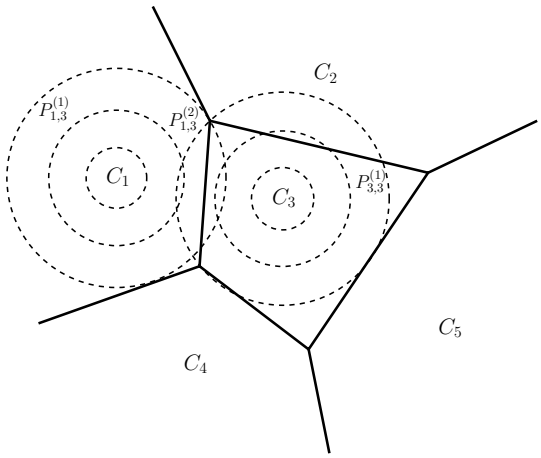


Figure 2: Examples of rings in the feature space, *i.e.*, on the surface of a hypersphere

Clusters:

$C_1 : \{C_{1,1} : P^{(1)}_{1,1}, P^{(2)}_{1,1}, ..., P^{(g)}_{1,1}\}, \{C_{1,2} : P^{(1)}_{1,2}, ..., P^{(g)}_{1,2}\}, ...$

$C_2 : \{C_{2,1} : P^{(1)}_{2,1}, P^{(2)}_{2,1}, ..., P^{(g)}_{2,1}\}, \{C_{2,2} : P^{(1)}_{2,1}, ..., P^{(g)}_{2,1}\}, ...$

...

Figure 3: Data structure for *iKernel*

We first present the indexing method (Section 5.2) assuming that the centroids and density-based clusters are properly determined. After that, we present the top-k query evaluation algorithm (Section 5.3). Finally, we present the density-based clustering algorithm that the indexing method uses (Section 5.4).

5.2 Index Construction

The index construction starts by determining clusters of instances and the reference points, *i.e.*, the centroids of the clusters. We will discuss the clustering algorithm in Section 5.4. The clusters are mutually exclusive and the cluster boundary resembles the intuition of boundaries in a voronoi diagram, as Figure 2 illustrates with an example of five clusters $C_1, .., C_5$ on the surface of a hypersphere.

After that, we start building rings from each centroid such that each ring contains g instances and the instances are sorted according to their distance orderings from the centroid. Since the distances of Eq.(4) among instances are unknown until the query time, the ring size is determined based on the number of instances it contains, not the distance. The orderings can be found by putting an arbitrary kernel parameter. We keep adding rings from each centroid until the rings include all the members in the cluster. The rings in each cluster only include the instances within the cluster so that each instance belongs to only one ring. For example, in Figure 2, points $P^{(1)}_{1,3}$ and $P^{(2)}_{1,3}$ belong to the third ring of C_1, and $P^{(1)}_{3,3}$ belongs to the third ring of C_3. We will have the data structured as Figure 3 where $C_{i,j}$ denotes the j^{th} ring of cluster C_i. Note that, *according to Lemma 2, the ring structure built on the ordering information is invariant to the kernel parameter that is given at query time.* The overall procedure of index construction is shown in Algorithm 1. We discuss the indexing complexity in Section 5.5.

5.3 Query Processing

With the index constructed, this section presents Algorithm *iKernel* finding top-k results for a given kernel ranking function F, with provable correctness and optimality.

Algorithm 1 Index construction

Input: m instances in the data space, kernel type
Output: an index constructed in the feature space

1. Cluster the instances based on their density and compute the centroids (locally peak points) of the clusters (Algorithm 3).
2. Construct rings from each centroid such that each ring contains g instances until the rings include all the members within the cluster.

To discuss the algorithm, we introduce a new notion of *minimal possible distance (MPD)*. $\text{MPD}_{C_{i,j}}$ is defined to be the minimal possible distance between a query point q and ring $C_{i,j}$. For example, in Figure 4(a), the minimal possible distance $\text{MPD}_{C_{2,3}}$ between the query point q and ring $C_{2,3}$ is the distance d_{C_2} between q and the centroid C_2 subtracted by the distance $d_{C_{2,3}}$ between C_2 and the farthest point in ring $C_{2,3}$.

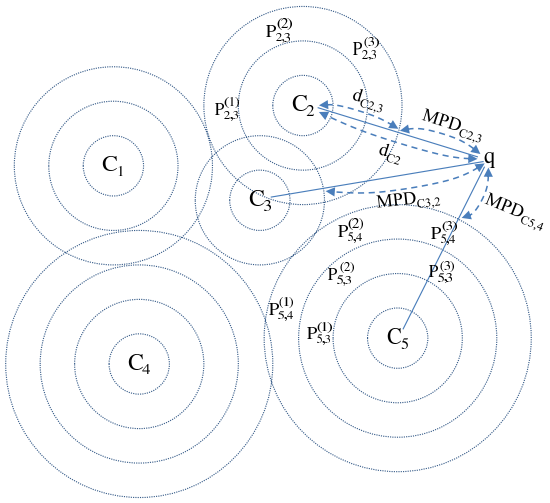
Based on this notion of the MPD, our top-k query processing algorithm runs as follows. When a query point q is given, we initialize a priority queue Q with the outermost ring of each cluster, in the ascending order of their MPDs between q. For example, in the case of Figure 4(a), Q will be initialized with five outermost rings and their MPDs, as shown in Figure 4(b). In an exceptional case where q belongs to some ring $C_{i,j}$, the MPD between q and the outmost ring of C_i will be set to a negative value and thus the outmost ring will be put on the top of Q.

We now illustrate how our top-k algorithm proceeds using an example in Figure 4(b) when $k = 2$. For simplicity, we represent the contents in the queue with ID only (though in our implementation, each item in the queue is either a ring or an instance annotated with MPD or distance to q respectively and the items are ordered by such distance in ascending order), which we represent in the table such that the leftmost item has the lowest distance. At each iteration, the top entry of Q is popped: At the first iteration, $C_{5,4}$ with the lowest MPD is popped. When a ring is popped, the distance from q to its instances, $P^{(1)}_{5,4}$, $P^{(2)}_{5,4}$, and $P^{(3)}_{5,4}$, and the MPD to its inner ring $C_{5,3}$, are computed and these instances are inserted back to Q, as the second row of the table in Figure 4(b) describes. At the second iteration, the top instance $P^{(3)}_{5,4}$ is added to output, which is guaranteed to be top-k, as the priority queue ensures that (a) all instances in the queue are farther from q and (b) also all rings have larger MPDs to q (which suggests that all the instances within these rings have even larger distance to q). At the third iteration, the top instance, ring $C_{5,3}$ is popped and its instances $P^{(1)}_{5,3}$, $P^{(2)}_{5,3}$, and $P^{(3)}_{5,3}$ are added to the queue. At the fourth iteration, the top instance, ring $C_{2,3}$ is popped and its instances $P^{(1)}_{2,3}$, $P^{(2)}_{2,3}$, and $P^{(3)}_{2,3}$ are added to the queue. Lastly, the top instance, $P^{(3)}_{5,3}$ is popped and returned as output. As top $k = 2$ instances are identified, Algorithm *iKernel* terminates. We describe this algorithm formally in Algorithm 2. The processing complexity is discussed in Section 5.5.

Note that in this example, only three rings are popped and evaluated to identify top-2 results. We prove that our algorithm is correct and no other correct algorithm can return correct results without accessing the rings accessed by our algorithm. More formally, Theorem 1 and Theorem 2 states the correctness and optimality respectively.

THEOREM 1 (CORRECTNESS). *Algorithm* iKernel *correctly returns the top-k answers.*

PROOF. We prove by contradiction. To contradict, we assume



(a)

step	output	top	updated Q (new items in **bold**)
0			$C_{5,4}, C_{2,3}, C_{3,2}, C_{1,3}, C_{4,4}$
1		$C_{5,4}$	$\mathbf{P^{(3)}_{5,4}}, \mathbf{C_{5,3}}, C_{2,3}, \mathbf{P^{(2)}_{5,4}}, \mathbf{P^{(1)}_{5,4}}, C_{3,2}, C_{1,3}, C_{4,4}$
2	$P^{(3)}_{5,4}$	$P^{(3)}_{5,4}$	$C_{5,3}, C_{2,3}, P^{(2)}_{5,4}, P^{(1)}_{5,4}, C_{3,2}, C_{1,3}, C_{4,4}$
3	$P^{(3)}_{5,4}$	$C_{5,3}$	$C_{2,3}, \mathbf{P^{(3)}_{5,3}}, \mathbf{C_{5,2}}, P^{(2)}_{5,4}, \mathbf{P^{(2)}_{5,3}}, P^{(1)}_{5,4}, \mathbf{P^{(1)}_{5,3}}, C_{3,2}, C_{1,3}, C_{4,4}$
4	$P^{(3)}_{5,4}$	$C_{2,3}$	$P^{(3)}_{5,3}, C_{5,2}, P^{(2)}_{5,4}, \mathbf{C_{2,2}}, P^{(2)}_{5,3}, \mathbf{P^{(3)}_{2,3}}, P^{(1)}_{5,4}, P^{(1)}_{5,3}, \mathbf{P^{(2)}_{2,3}}, \mathbf{P^{(1)}_{2,3}}, C_{3,2}, C_{1,3}, C_{4,4}$
5	$P^{(3)}_{5,4}, P^{(3)}_{5,3}$	$P^{(3)}_{5,3}$	$C_{5,2}, P^{(2)}_{5,4}, C_{2,2}, P^{(2)}_{5,3}, P^{(3)}_{2,3}, P^{(1)}_{5,4}, P^{(1)}_{5,3}, P^{(2)}_{2,3}, P^{(1)}_{2,3}, C_{3,2}, C_{1,3}, C_{4,4}$

(b)

Figure 4: Top-2 query processing

Algorithm 2 Query processing

Input: an index, a kernel ranking function F
Output: top-k result set $output$
/* intermediate data structure
Q: a priority queue with elements of (RingID, MPD) or (InstanceID, distance)
*/

1. Compute the MPDs between the query point and the outmost rings of all clusters, and put them into Q. The rings with shorter MPDs go on the top of the queue.
2. Pop the top item.
   2.1. If the item is an instance,
      2.1.1. append the item to $output$
   2.2. Else,
      2.2.1. Evaluate the instances in $C_{i,j}$ and insert them into Q.
      2.2.2. Push $C_{i,j-1}$ into Q.
3. Repeat 2 until $|output| \geq k$.
4. Return output

iKernel leaves out some top-k u (which belongs to ring C_u) from the results K. Naturally, distance between u and q is no larger than that between the k^{th} nearest instance v to q, *i.e.*, $d(u,q) \leq d(v,q)$ (1). As Algorithm *iKernel* accesses all rings with MPD no larger than $d(v,q)$, *i.e.*, $MPD_{C_u} > d(v,q)$(2). Combining (1) and (2), we get $d(u,q) < MPD_{C_u}$, which contradicts the definition of MPD. □

THEOREM 2 (OPTIMALITY). *Given the ring index, Algorithm* iKernel *accesses the minimal number of rings.*

PROOF. We prove by contradiction. Assume another algorithm A returns the correct top-k results, without accessing some ring C accessed by *iKernel*. Algorithm *iKernel* assures that MPD_C is no larger than the distance of top-k results. This suggests that if some instance in ring C has the distance of MPD_C to q, this instance will be included in the results of *iKernel*, but not in those of A, which incurs contradiction. □

5.4 Clustering to Maximize the Pruning Ratio

So far, we have discussed *iKernel*, for the given clusters, but recall that, the quality of such clusters greatly impacts on the pruning ratio in the query processing. Intuitively, to ensure high pruning ratio, clusters should ensure that MPD (minimal possible distance) tightly bounds true minimal distances between the query point and the cluster, for which clusters should be centered at the dense regions.

To illustrate, Figure 5(a) and (b) contrast cluster centroids placed at dense regions and random regions, respectively. In case of random centroids, data instances in each ring are highly skewed and also the rings between clusters are highly overlapped. That is, MPDs for these rings are much lower than the true minimal distance, which makes these rings less likely to be pruned.

The key challenge in finding such density-based clusters in the feature space is that we only have the information about pairwise distances without being able to see the actual points. Typical clustering methods such as the K-mean algorithm are not applicable since they do not find locally density maximum points as the centroids and also it is impossible to compute the mean points without being able to see the points.

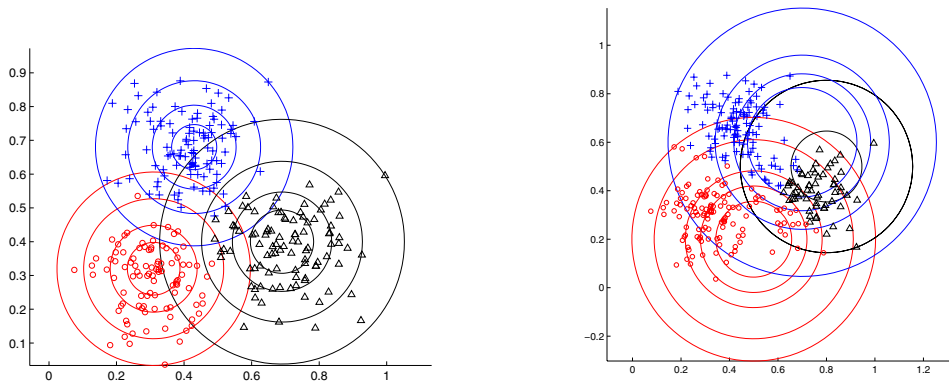
Our approach is to estimate the density of each point $\vec{x}_i$ by summing the influence of other points to $\vec{x}_i$. Specifically, the density $I(\vec{x}_i)$ of an instance $\vec{x}_i$ is estimated as:

$$I(\vec{x}_i) = \sum_j exp(-\gamma \cdot D(\vec{x}_i, \vec{x}_j)^2) \quad (5)$$

Parameter γ controls the degree of the influence. This density estimation is indeed equal to the kernel density estimation except that our distance function returns the feature space distance of Eq.(4). Thus, for RBF kernel,

$$I(\vec{x}_i) = \sum_j exp(-\gamma(cos^{-1}(exp(-s||\vec{x}_i - \vec{x}_j||^2)))^2) \quad (6)$$

Note that, we have an additional kernel parameter, *e.g.*, s in the RBF. Since the kernel parameter controls the distance scale between two points, it plays the same role as γ. Thus, we only need to adjust one of either parameters while fixing the other.



(a) Rings with locally density maximum centroids

(b) Rings with random centroids

Figure 5: Rings with random or locally density maximum centroids

Once we compute the density of every point $\vec{x}$, we need to find local density maximum points (or local peaks) in the feature space. However, since actual coordinates of points are unknown in the feature space, it is hard to determine whether current points are locally maximum, minimum, or on the slope. Thus, we approximate the centroids by choosing them from the highest density points that are at least $2r$ away from other centroids. We fixed the radius of clusters into r. Once we determine the centroids, we associate each point with the closest centroid. Our clustering algorithm has two parameters, γ and r, which determine the number of clusters. These parameters can be tuned experimentally using synthetic queries to maximize the pruning ratio. Once they are tuned for a data set, it does not have to be tuned for each query. The clustering algorithm is described in Algorithm 3.

Algorithm 3 Clustering

Input: data instances, kernel $\mathcal{K}$, radius r
Output: an index

1. Compute the density scores of instances in the feature space (Eq.(5)).
2. Sort the instances according to the density scores.
3. Set centroids $C = \emptyset$.
4. Loop for each instance $\vec{x}$ from the highest density.
   4.1. If the distance between $\vec{x}$ and all the centroids in C is larger than $2r$,
      4.1.1. $C = C \cup \vec{x}$
5. Associate each instance with the closest centroid.

5.5 Discussions

Cost for indexing: The index construction time is dominated by the clustering time, as computing the density functions Eq.(5) for all the data takes $\mathcal{O}(m^2)$ where m is the number of instances. However, when randomly selecting the centroids, the index can be constructed in $\mathcal{O}(m \cdot c)$ where c is the number of centroids. We only used the random centroids on large data sets in our experiments due to the high complexity of clustering, and *iKernel* still produced 1∼5% of evaluation ratio overall. Moreover, the index construction is an offline process which does not affect the response time of query processing. We experimentally report in Section 6.5 that the index size and construction time of *iKernel* is substantially lower than that of the approximation method.

Cost for processing: To process a query on *iKernel*, the MPDs between the query point and all clusters must be first computed. Thus, the cost for query processing is $c + p * g$ where c is the number of clusters, p is the number of rings examined to process a query, and g is the number of instances in each ring.

Since *iKernel* is composed of a set of centroids and rings sorted by their distances to the centroids, the rings in each cluster can be sequntially placed on the disc. To minimize random access in indexing and query processing, we can store the centroids and their MPDs in memory or sequential blocks in disc, and lay out the rings corresponding to each centroid sequentially in disc. Then, locating rings of different clusters may incur random accesses but examining rings within the same clusters only involves sequential accesses.

Assume the centroids are stored in l blocks, and the g instances in each ring are stored in one block in disc. If the number of rings examined for processing a query is p, the query processing involves $l + p$ block accesses. l blocks are sequential accessed, and p blocks may or may not be sequentially accessed.

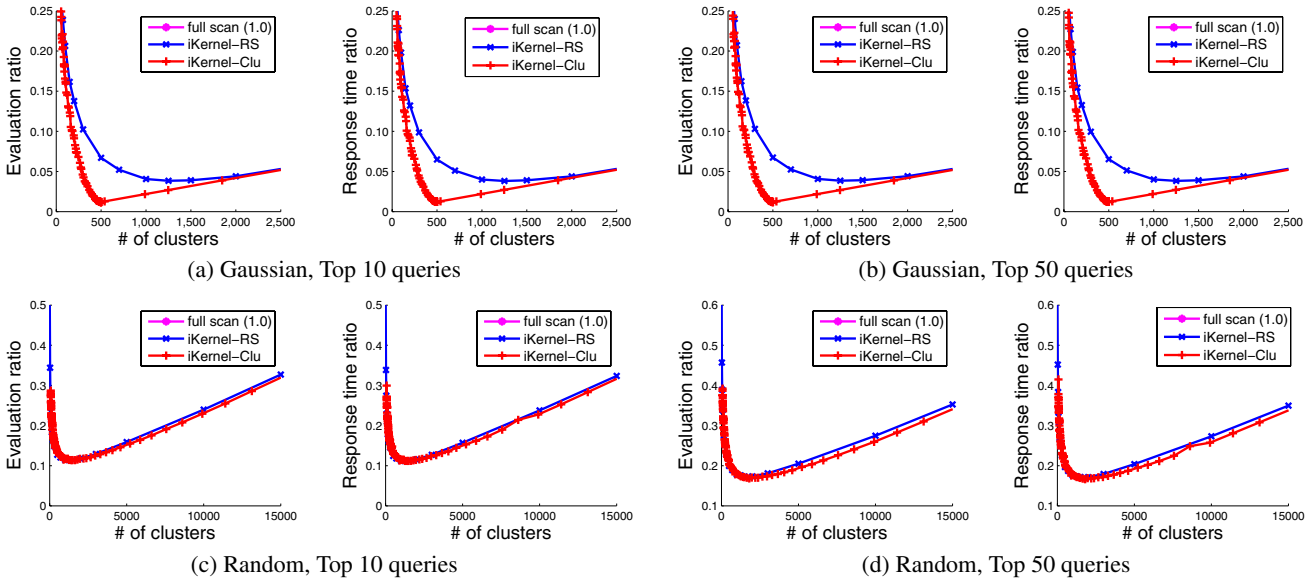
Insertion and Deletion: New instances are inserted to the nearest rings. Inserting an instance requires identifying (1) the cluster by finding the nearest centroid and (2) the proper ring of the cluster, which would take $c + r + g$ where c, r, and g are the number of clusters, rings for a cluster, and instances within a ring respectively. Deleting process is also similar.

As instances are inserted or deleted from the index, some rings may have many instances while other rings have few, which could incur skewed rings and imbalanced clusters. Since we fixed the size of each ring into g, when each ring has more than $2g$, we could split the ring into two adjacent rings. Similarly, we could merge two adjacent rings into one ring when they have few instances ($< 0.5g$).

However, according to our experiments in Section 6, even without the split or merge of rings, insertion or deletion operations make little impact in the pruning performance. Thus, we can also simply reconstruct the index if it starts making noticeable performance drop, which would take $\mathcal{O}(m \cdot c)$ with random centroids.

6. EXPERIMENT

This section evaluates the effectiveness of our methods on synthetic and real-world data sets. To fully show the usefulness of our methods, we have measured the performance of *iKernel* in var-



(a) Gaussian, Top 10 queries

(b) Gaussian, Top 50 queries

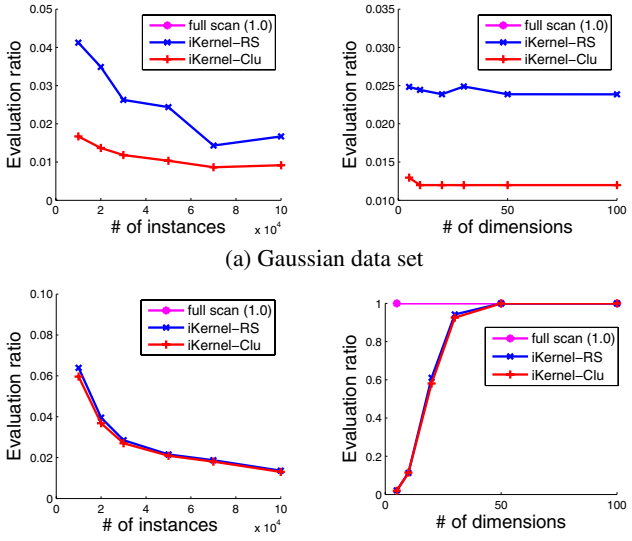
(c) Random, Top 10 queries

(d) Random, Top 50 queries

Figure 6: Experiments on synthetic data: comparison of evaluation and response time.

ious situations such as changing kernel parameters and inserting (deleting) instances into (from) the index. We also compared our methods with the existing approximation method, KDX. Our experiments were done on a linux machine with two quadcore CPUs (2.27GHz) and 24G memory.

6.1 Synthetic Data



(a) Gaussian data set

(b) Random data set

Figure 7: Experiments on synthetic data

We first performed experiments on two types of synthetic data – Random and Gaussian. The two types of synthetic data sets are generated as follows.

- Random: we randomly generated 50k instances of 10 dimensions such that each attribute is a real value between 0 and 1.
- Gaussian: we generated 50k instances of 10 dimensions using the Gaussian distributions with random means and fixed standard deviations. That is, we randomly picked 500 centroids from each of which 100 points are generated using the Gaussian distributions.

For each data set, we randomly generated 10 queries, evaluated the performance of *iKernel* over the queries, and averaged the performance results. The queries were generaed using RankSVM with RBF kernels by randomly picking 25 positive and 25 negative points as training data. The 25 positive points were selected from the nearest neighbors of a randomly selected point x, and the 25 negative points were selected from the farthest points of x. When training, the kernel parameter s and the penalty term C were set to $\frac{0.01}{\sqrt{\#\text{dimensions}}}$ and 0.01, repsectively. We will show the performance change of *iKernel* on different kernel settings later. We evaluated *iKernel* with random and density-based clustered centroids, which we denote as:

- "iKernel-Clu": *iKernel* with centroids determined by our proposed clustering method
- "iKernel-RS": *iKernel* with randomly sampled centroids

As iKernel-RS randomly selects centroids, the clustering results may be different at each run. For statistical significance, we have experimented 10 runs and averaged the results.

Figure 6 shows the evaluation ratio and response time ratio according to the number of clusters. We varied the number of clusters by reducing the radius r in Algorithm 3. Figure 6(a) and (b) show the results on the Gaussian data set. iKernel-Clu showed the best performance, *i.e.*, the lowest number of evaluations and the shortest response time, with 500 centroids determined. The evaluation and response time ratios saved by *iKernel* are very close to each other, which implies the instance evaluation cost dominates the overall response time. It is because the evaluation of the kernel ranking function involves the nonnegligible cost of scanning the support vectors. Observe also that both iKernel-Clu and iKernel-RS, *demonstrate consistently high performance, incurring less than 3% of full scan cost.*

Dataset	# of evaluations			response time (sec.)		
	iKernel	full scan	evaluation ratio	*iKernel*	full scan	response time ratio
Covtype	6071.052	581012	0.010	0.309	24.506	0.013
Iris	3441.532	500150	0.007	0.024	3.568	0.007
Ijcnn	4632.087	126701	0.037	0.087	2.468	0.035
Vehicle	5262.824	98528	0.053	0.188	3.600	0.052
Statlog-shuttle	232.100	57999	0.004	0.002	0.600	0.004
Handwritten digit	1476.995	10992	0.134	0.407	3.088	0.132
Abalone	133.300	4177	0.032	0.015	0.507	0.030
Mg	211.899	1385	0.153	0.017	0.110	0.155
Fourclass	63.700	862	0.074	0.005	0.062	0.081

Table 1: The number of evaluations and response time of *iKernel* on real-world data sets

Figure 6(c) and (d) show the results on the Random data set. Note that, since the instances here were randomly generated without forming any clusters, the peak performance was obtained when around 1k clusters were used, which means each cluster contains only about 50 points on average. iKernel-Clu and iKernel-RS demonstrated comparable performance showing around 10% of evaluation ratio for top-10 queries. Compared to the performance observed for clustered data (in Figure 6(a) and (b)), *iKernel is less effective on randomly distributed data.*

Next, we examined the performance of *iKernel* with varying the number of dimensions and instances on the Gaussian and Random data sets shown in Figure 7(a) and (b) respectively. When the data were well-clustered (Figure 7(a)), the relative effectiveness of *iKernel* growed as data size increased and remained unchanged as dimensionality increased. However, when the data were not well-clustered (Figure 7(b)), *iKernel* was effective only on low-dimensional data and it ended up fully scanning when the dimenion was 50 or higher. The results of these experiments tell that *how well the data set is clustered is critical to the performance of iKernel*.

6.2 Real-world Data

This section uses real-world data sets obtained from the UCI machine learning repository. We picked nine data sets of varying sizes. The data sets described in Table 2 are sorted according to their sizes in descending order.

Name	# of instances	# of attributes
Covtype	581,012	54
Iris	500,150	4
Ijcnn	126,701	22
Vehicle	98,528	50
Statlog-shuttle	57,999	9
Handwritten digit	10,992	16
Abalone	4,177	8
Mg	1,385	6
Fourclass	862	2

Table 2: Real-world data sets

Since the Iris data set is too small, we increased its size from 150 to 500,150 by performing the SMOTE algorithm [5]. Queries were generated using RankSVM with the same settings described in Section 6.1. We fixed the density parameter $\gamma = \frac{10.0}{cos^{-1}(0)}$ for all data sets. We performed top-10 queries, $k = 10$.

Table 1 shows the number of evaluations and response time of *iKernel* compared to those of full scanning. Note that *iKernel* consistently showed significant performance enhancement on all data sets. *iKernel* determined top-k results by seeing 1% or less data on very large data sets (> 500k) and 7% on the other data sets except *handwritten digit* and *mg*. Even for them, *iKernel* only saw less than about 15%. Response time ratio showed similar results, as instance evaluation ratio dominated the performance. From these results, we conclude that *iKernel is highly effective on various data sets.*

Figure 8 shows the evaluation ratios with varying number of clusters. Note that, for the first four data sets, we only used the index with random centroids, as their sizes were too big to run the clustering. *iKernel* with random centroids were still highly effective. Note that *we fixed the clustering parameters for all the data sets, but a data-dependent parameter tuning could improve the performance even further.*

$\log_2 C$	σ^2			
	30	40	50	60
-4	0.0372	0.0386	0.0391	0.0386
-3	0.0331	0.0360	0.0366	0.0371
-2	0.0326	0.0334	0.0327	0.0335
-1	0.0305	0.0307	0.0331	0.0304
0	0.0314	0.0316	0.0313	0.0338
1	0.0309	0.0282	0.0310	0.0293
2	0.0298	0.0305	0.0315	0.0312
3	0.0287	0.0351	0.0352	0.0297
4	0.0283	0.0331	0.0316	0.0323

Table 3: Ijcnn: Evaluation ratios on various kernel parameters

$\log_2 C$	σ^2			
	30	40	50	60
-4	0.0180	0.0172	0.0173	0.0169
-3	0.0180	0.0176	0.0172	0.0177
-2	0.0197	0.0178	0.0178	0.0207
-1	0.0197	0.0178	0.0174	0.0175
0	0.0197	0.0178	0.0174	0.0175
1	0.0197	0.0178	0.0174	0.0175
2	0.0197	0.0178	0.0174	0.0175
3	0.0197	0.0178	0.0174	0.0175
4	0.0197	0.0178	0.0174	0.0175

Table 4: Covtype: Evaluation ratios on various kernel parameters

6.3 Changing Kernel Parameters

So far, experiments were conducted over queries generated using fixed kernel parameters, $s = \frac{1}{2\sigma^2} = \frac{0.01}{\sqrt{\#\text{dimensions}}}$ and $C = 0.01$. However, in reality, users generate queries with various kernel pa-

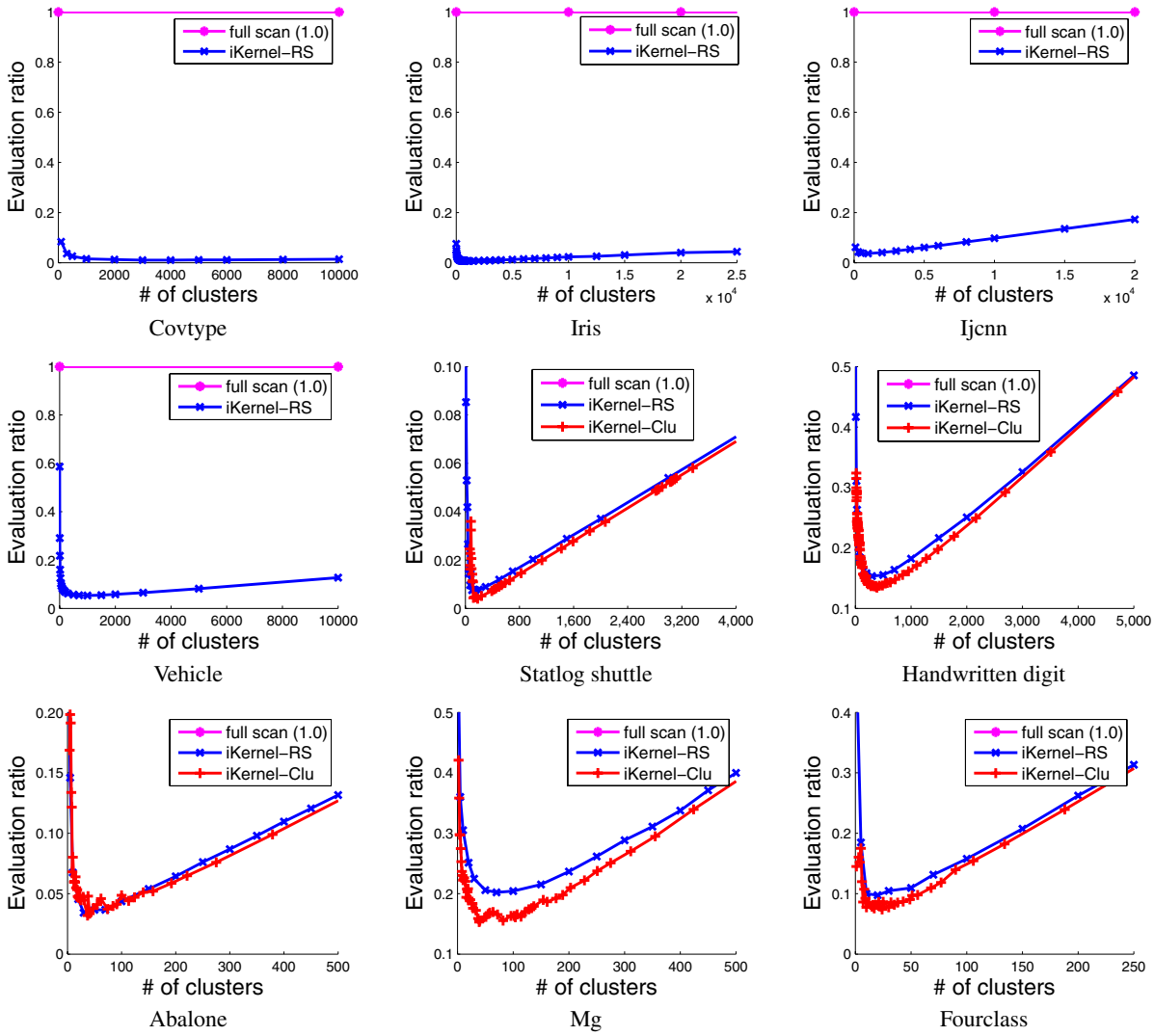


Figure 8: Experiments on Real-world Data: Comparison of evaluation ratio.

rameters to fit the model to their data. To identify the effectiveness of our methods on these situations, we measured the performance of *iKernel* on various queries generated using different kernel parameters.

To reflect the real situation as much as possible, we selected two large data sets *covtype* and *ijcnn*. For each of them, *iKernel* index is generated with fixed number of clusters. While fixing the index, we generated queries with various kernel parameters. The kernel parameters have value range of $\sigma^2 = 30, 40, 50, 60$ and $C = 2^{-4}, \ldots, 2^4$. For each pair of the kernel parameters, 10 queries were generated and the results were averaged over them.

Table 3 and 4 show the evaluation ratios according to the kernel parameter changes. The performance of *iKernel* was not almost affected by kernel parameter changes. The evaluation ratios changed less than 1%. *iKernel* is *still highly effective on various queries with different kernel parameters.*

6.4 Inserting and Deleting Instances

To evaluate the performance of *iKernel* in dynamic environments where instances are inserted or deleted, we measured the evaluation ratios of *iKernel* as we inserted (deleted) instances into (from) the index for the *ijcnn* data set.

We first measured the performance when instances were inserted. Initially, we had 50k instances and created clusters for them. We inserted 10k instances at a time and measured the performance of *iKernel* on the queries generated from the cumulative instances up to that time with the same settings described in Section 6.2. Total of 60k instances were inserted.

Figure 9 shows that the evaluation ratios were increased as the number of inserted instances increased except when the number of initial clusters was 2000. Since the distances between the centroids and the farthest instances from the clusters increased as we inserted instances, the overlapping area between clusters would be increased large enough to increase the evaluation ratios. However, if we picked more instances as centroids in advance, the overlapping area would increase small and the number of clusters to see to find the top-k results would not increase much relative to the inserted number of instances, resulting in less evaluation ratios. In all cases, the evaluation ratios increased at most about 1.5%.

Next we measured the performance when instances were deleted. The initial number of instance was 120k and we removed 10k in-

stances at a time, the opposite of insertion case. The total number of deleted instances was 60k.

Figure 10 shows that the evaluation ratios decreased in all cases. This is because in the opposite of inserting, by removals of instances, the overlapping area among the clusters would be getting smaller.

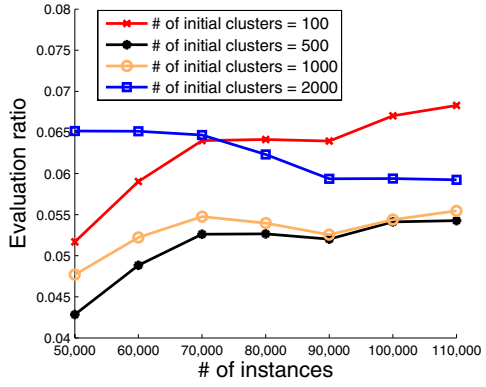


Figure 9: Ijcnn: Evaluation ratios according to # of insertions

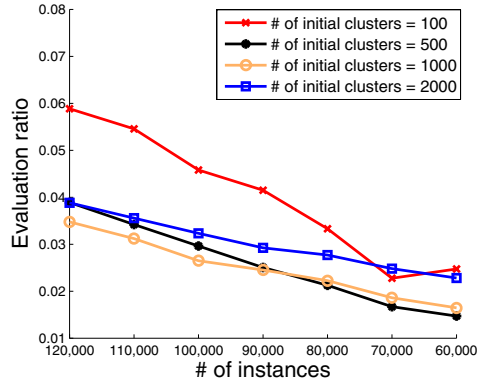


Figure 10: Ijcnn: Evaluation ratios according to # of deletions

6.5 Performance Comparison with KDX

Motivated by the high effectiveness of our methods, we compared the performance of our methods with the performance of the existing approximation SVM indexing method, KDX [15]. We first compared the performance on the settings described in [15] and then on our settings described in Section 6.2. We also compared the index size and construction time of the two methods.

Among the data sets in [15], we selected two large data sets, *ijcnn* and *vehicle*, which are publicly available and are large enough to reflect the real situations. The two data sets were preprocessed in the same way described in [15] and the queries were also generated in the same way.

Data	Evaluation ratios		Recall		Discrepancy	
	iKernel	KDX	*iKernel*	KDX	*iKernel*	KDX
Ijcnn	0.102	0.241	1.000	0.650	0.0	0.250
Vehicle	0.175	0.122	1.000	0.783	0.0	0.120

Table 5: Performance comparison between iKernel and KDX on KDX experimental settings

Table 5 compares the results of *iKernel* and KDX on these settings. The results of KDX are taken from the results reported in [15]. Note that, as KDX is an approximation method, the authors of KDX reported *recall* and *evaluation ratios* until that recall is obtained. The authors of KDX also reported *discrepancy* which is the relative difference of averaged query values (*i.e.,* evaluated values of top-k results) between the true results and the approximated results. The larger the discrepancy is, the more different the query values are. In contrast, as our methods are *exact methods*, ours always returns exact results with recall 1.0 and discrepancy 0. Our methods showed comparable or better evaluation ratio even with the perfect recall.

Encouraged by the high effectiveness of the previous results, we continued the comparison on our settings described in Section 6.2. For the fairness of comparison, we tuned the parameters of KDX[1] to find the best performance on each data set. For the repeatability of the experiments, we report the parameters we tried in Table 6. Note that among the results obtained by varying the parameters described in Table 6, we report the results which showed the highest recall values in Table 7.

Data	*g* (# of data in a ring)	*max # of unchanges*
Covtype Iris Ijcnn Statlog-shuttle	500 - 3000 (incremented by 250)	1, 3, 5, 10, 15, 20, 25, 30
Handwritten digit	50 - 500 (incremented by 50)	1, 3, 5, 10, 15, 20
Abalone Mg Fourclass	10, 20, 30, 50, 100, 150, 200	1, 3, 5, 10

Table 6: KDX parameters we tried

Dataset	Evaluation ratios		Recall (KDX)	Discrepancy (KDX)
	iKernel	KDX		
Covtype	0.010	0.001	0.114	0.374
Iris	0.007	0.001	0.381	0.199
Ijcnn	0.037	0.004	0.026	0.370
Vehicle	0.053	0.002	0.132	0.370
Statlog	0.004	0.002	0.598	0.040
Handwritten	0.134	0.057	0.143	0.482
Abalone	0.032	0.017	1.000	0.000
Mg	0.153	0.352	1.000	0.000
Fourclass	0.074	0.144	1.000	0.000

Table 7: Performance comparison between iKernel and KDX on our experimental settings

From Table 7, we see that KDX returned exact results, *recall=1.0*, on small data sets (*abalone, mg,* and *fourclass*), but the evaluation ratio was high, more than 35% for *mg* data set. For large data sets, KDX showed very low evaluation ratios, less than 0.4% of evaluation ratio, but the recalls were significantly reduced and the discrepancies were also high. Our methods, on the other hand, consistently showed low evaluation ratios while returning the exact results with perfect recall.

Finally, we compared the index size and construction time of *iKernel* and KDX. Table 8 shows that our methods constructed indexes much faster, as ours required much smaller storage than KDX

[1]We implemented the KDX algorithm according to the pseudo code given in the paper, which is straightforward.

for large data sets. These are consistent results with our analysis that our index construction time complexity is $\mathcal{O}(mc)$ whereas the time complexity of KDX is $\mathcal{O}(m^2)$. Also the storage time complexity of our methods is $\mathcal{O}(m)$ whereas the time complexity of KDX is $\mathcal{O}(mg)$ and g is not trivially small, thus the results showed that KDX required very large storage. These large storage requirement may incur larger number of page accesses during operation.

Dataset	*iKernel*		KDX	
	Time	Size	Time	Size
Covtype	749.686	382M	64931.077	6.8G
Iris	58.257	42M	13266.685	7.5G
Ijcnn	26.882	37M	1788.108	5.3G
Vehicle	38.950	61M	1921.981	4.5G
Statlog-shuttle	501.725	36M	298.589	2.4G
Handwritten digit	17.209	2.5M	9.862	11M
Abalone	1.825	555K	1.212	1.1M
Mg	0.191	152K	0.112	314K
Fourclass	0.064	54K	0.040	186K

Table 8: Index construction time and size of iKernel and KDX on our experimetal settings

In summary, *iKernel* is highly effective on large data sets producing less than 1$\sim$5% evaluation ratio overall with perfect recall while KDX produces very low recall on large data sets. The indexing size and construction time of *iKernel* is significantly better than KDX, and the maintenance for insertion and deletion operations is also inexpensive making little impact on the performance of *iKernel*.

7. CONCLUSIONS

This paper proposes an indexing method and top-k processing algorithm for the ranking functions learned by SVMs, *i.e.*, the non-linear kernel ranking functions. Key challenges in developing indexes for the kernel ranking function are that the data instances are only defined with respect to the "parameterized" pairwise distances in the feature space and thus their absolute feature values are invisible and the real pairwise distances are determined at the query time. Our proposed method, *iKernel*, builds an index structure in the feature space using only the information of ordering of distances that is invariant to the kernel parameters thus unchanging with the queries. According to our experiments, we observe that *iKernel* consistently outperforms full scanning and is especially more effective when the given data is *large or well-clustered*.

8. REFERENCES

[1] T. Bozkaya and M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transaction on Database Systems*, 2002.

[2] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *Proc. Int. Conf. Data Engineering (ICDE'02)*, 2002.

[3] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

[4] E. Chang and S. Tong. Support vector machine active learning for image retrieval. In *ACM Int. Conf. Multimedia (MM'01)*, 2001.

[5] N. Chawla, K. Bowyer, L. Hall, and W. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 2002.

[6] N. Christianini and J. Shawe-Taylor. *An Introduction to support vector machines and other kernel-based learning methods*. Cambridge University Press, 2000.

[7] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. Int. Conf. Very Large Databases (VLDB'97)*, 1997.

[8] R. Fagin, A. Lote, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems (PODS'01)*, 2001.

[9] U. Guentzer, W. Balke, and W. Kiessling. Optimizing multi-feature queries in image databases. In *Proc. Int. Conf. Very Large Databases (VLDB'00)*, 2000.

[10] S.-W. Hwang and K. C.-C. Chang. Optimizing access cost for top-k queries over web sources: A unified cost-based approach. In *Proc. Int. Conf. Data Engineering (ICDE'05)*, 2005.

[11] H. Jagadish, B. Ooi, K. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Transaction on Database Systems*, 2005.

[12] D. Keim. Tutorial on high-dimensional index structures: Database support for next decades applications. In *Proc. Int. Conf. Data Engineering (ICDE'00)*, 2000.

[13] N. Panda and E. Chang. Exploiting geometric property for support vector machine indexing. In *SIAM Int. Conf. Data Mining (SDM'05)*, 2005.

[14] N. Panda and E. Chang. Efficient top-k hyperplane query processing for multimedia information retrieval. In *ACM Int. Conf. Multimedia (MM'06)*, 2006.

[15] N. Panda and E. Chang. Kdx: An indexer for support vector machines. *IEEE Transactions on Knowledge and Data Engineering*, 2006.

[16] J. Peng and D. Heisterkamp. Kernel indexing for relevance feedback image retrieval. In *Proc. Int. Conf. Image Processing (ICIP'03)*, 2003.

[17] V. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, 1998.

[18] D. Xin, J. Han, and K. Chang. Progressive and selective merge: Computing top-k with ad-hoc ranking functions. In *Proc. ACM Int. Conf. Management of Data (SIGMOD'07)*, 2007.

[19] H. Yu. SVM selective sampling for ranking with application to data retrieval. In *Proc. Int. Conf. Knowledge Discovery and Data Mining (KDD'05)*, 2005.

[20] H. Yu. Selective sampling techniques for feedback-based data retrieval. *Data Mining and Knowledge Discovery*, 2010.

[21] H. Yu, S. Hwang, and K. C.-C. Chang. RankFP: A framework for supporting rank formulation and processing. In *Proc. Int. Conf. Data Engineering (ICDE'05)*, 2005.

[22] H. Yu, S.-W. Hwang, and K. C.-C. Chang. Enabling soft queries for data retrieval. *Information Systems*, 2007.

[23] H. Yu, T. Kim, J. Oh, I. Ko, S. Kim, and W. Han. Enabling multi-level relevance feedback on pubmed by integrating rank learning into dbms. *BMC Bioinformatics*, 2010.

[24] Z. Zhang, S.-W. Hwang, K. C.-C. Chang, M. Wang, C. A. Lang, and Y.-C. Chang. Boolean + ranking: Querying a database by k-constrained optimization. In *Proc. ACM Int. Conf. Management of Data (SIGMOD'06)*, 2006.