Maximum inner-product search using cone trees

Article · August 2012

DOI: 10.1145/2339530.2339677

2 authors, including:

Alexander Gray
Skytree Inc.
132 PUBLICATIONS **2,899** CITATIONS

SEE PROFILE

Maximum Inner-Product Search Using Cone Trees

Parikshit Ram
School of Computational Science & Engineering
Georgia Institute of Technology
p.ram@gatech.edu

Alexander G. Gray
School of Computational Science & Engineering
Georgia Institute of Technology
agray@cc.gatech.edu

ABSTRACT

The problem of efficiently finding the best match for a query in a given set with respect to the Euclidean distance or the cosine similarity has been extensively studied. However, the closely related problem of efficiently finding the best match with respect to the inner-product has never been explored in the general setting to the best of our knowledge. In this paper we consider this problem and contrast it with the previous problems considered. First, we propose a general branch-and-bound algorithm based on a (single) tree data structure. Subsequently, we present a dual-tree algorithm for the case where there are multiple queries. Our proposed branch-and-bound algorithms are based on novel inner-product bounds. Finally we present a new data structure, the cone tree, for increasing the efficiency of the dual-tree algorithm. We evaluate our proposed algorithms on a variety of data sets from various applications, and exhibit up to five orders of magnitude improvement in query time over the naive search technique in some cases.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Trees; G.4 [**Mathematical Software**]: Algorithm design and analysis; H.2.8 [**Database Management**]: Database Applications—*Data mining*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

General Terms

Algorithms, Design

Keywords

Metric trees, cone trees, dual-tree branch-and-bound

1. INTRODUCTION

In this paper, we consider the problem of efficiently finding the best match for a query from a given set of points with respect to the inner-product similarity. We focus on improving the efficiency of this search. Formally, we consider the following problem:

KDD'12, August 12–16, 2012, Beijing, China.

Maximum inner-product search. For a given set of N points $S \subset \mathbb{R}^D$ and a query $q \in \mathbb{R}^D$, efficiently find a point $p \in S$ such that:

$$\langle q, p \rangle = \max_{r \in S} \langle q, r \rangle. \quad (1)$$

At first glance, this problem appears to be very similar to much existing work in literature. Efficiently finding the best match with respect to the Euclidean (or more generally L_p) distance is the widely studied problem of fast nearest-neighbor search in metric spaces [9]. Efficient retrieval of the best match with respect to the cosine similarity has been researched in the field of text mining and information retrieval [1]. But as we will explain in the next section, the maximum inner-product search is not only different from these aforementioned tasks, but also arguably harder.

1.1 Applications

An obvious application of maximum inner-product search stems out of the widely successful matrix-factorization framework in recommender system challenges like the "Netflix prize" [22, 21, 2]. The matrix-factorization results in accurate representation of the available data in terms of user vectors and items vectors (examples for items would be movies or music). In this setting, the preference of a user for an item is the inner-product between the corresponding user's vector and the item's vector[1]. The retrieval of recommendations for a user is equivalent to maximum inner-product search with the user as the query and the items as the reference set. Linear scan of the items are usually employed to find the best recommendations. An efficient search algorithm would make the retrieval of recommendations in the matrix-factorization framework scalable to larger systems.

The usual document retrieval tasks use the cosine similarity to match documents. However, in certain settings [11], the documents are represented as (not necessarily normalized) vectors and the inner-product between these vectors represent their mutual similarity. In this case, unless the vectors are normalized to have the same length, document matching using the cosine similarity [1] might make the algorithm scalable at the cost of returning inaccurate solutions since the inner-product is not the same as the cosine similarity (we will discuss this further in Section 2).

There is a similar problem known as the the max-kernel operation: for a given set of points S and a query q and a kernel function $\mathcal{K}(\cdot, \cdot)$, the task is to find the point $p \in S$ with the maximum value of $\mathcal{K}(q, p)$ over the set S. This

[1]The preference is actually the inner-product between the user and the item vector plus a item bias term. But this can be reduced to an inner-product by appending the user vector with 1 and the item vector with the item bias.

problem is widely used in maximum-a-posteriori inference [20] in machine learning, and for image matching [23] in computer vision. If the kernel function can be explicitly represented in the form a function $\varphi(\cdot)$ such that $\mathcal{K}(q,p) = \langle\varphi(q),\varphi(r)\rangle$, then this problem reduces to maximum inner-product search after all the points in the set S and the query q is transformed into the φ-space.

1.2 This Paper

In this paper, we propose two tree-based branch-and-bound algorithms along with a new data structure to solve this problem. In Section 2, we contrast this problem to the more familiar problems of nearest-neighbor search in metric spaces and best matches with respect to the cosine similarity. In Section 3, we propose a simple branch-and-bound algorithm using existing ball tree data structure [28] and a novel bound. In the following section (Section 4), we address the situation where there are multiple queries on the same set of points and propose a dual-tree branch-and-bound algorithm. In Section 5, we present a new data structure, the*cone trees*, to index the queries for the dual-tree algorithm. These structures take advantage of novel inner-product bounds which are tighter than those that can be achieved with traditional ball trees. The proposed algorithms are evaluated for their efficiency over a variety of data sets in Section 6. Section 7 demonstrates how the proposed algorithms can be applied to the max-kernel operation with general kernel functions without any explicit representation of the points in the φ-space. In the final section, we provide our conclusions along with possible future directions for this work.

2. MAXIMUM INNER-PRODUCT SEARCH

Numerous techniques exists for nearest-neighbor search in Euclidean metric space (see surveys like [9]). Large scale best matching algorithms have also been developed for the cosine-similarity measure [1], with a lot of focus on text data. The problem of nearest-neighbor search (in metric space) has been solved approximately with the widely popular *locality-sensitive hashing* (LSH) [14, 18]. LSH has been extended to other forms of similarity functions (as opposed to the distance as a dissimilarity function) like the cosine similarity [7][2]. The approximate max-kernel operations can also be solved efficiently with LSH under certain conditions on the kernel function. Dimension reduction [30] and dual-tree algorithms [20] have also been used to solve the approximate max-kernel operation efficiently.

2.1 How is maximum inner-product search different from existing problems?

Here we explain why the maximum inner-product search is different from these existing search problems. Hence techniques applied to these problems (like LSH) cannot be directly applied to this problem.

Nearest-neighbor search in Euclidean space. This involves finding a point $p \in S$ for a query q such that:

$$\begin{aligned} p &= \arg\min_{r\in S} \|q-r\|_2^2 = \arg\max_{r\in S}\left(\langle q,r\rangle - \frac{\|r\|_2^2}{2}\right) \\ &\neq \arg\max_{r\in S}\langle q,r\rangle \text{ (unless } \|r\|_2^2 = k\ \forall\ r\in S). \end{aligned}$$

[2]An important thing to note here is that the similarity function used in Charikar et.al.[7] is not exactly the cosine-similarity. The distance between two points p and q was measured by θ/π, where θ is the angle made the two points at the origin, making the similarity function $\left(1-\frac{\theta}{\pi}\right)$. This similarity function has a direct correspondence to the cosine similarity.

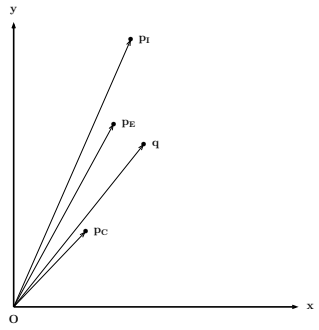

Figure 1: Best matches: For a given query q, p_C, p_E and p_I denote the different best match candidates with respect to the cosine similarity, the Euclidean distance and the inner-product respectively.

Hence, if the norms of all the points in S are normalized to have the same length, then maximum inner-product search is equivalent to nearest-neighbor search in Euclidean metric space. However, without this restriction, the two problems can have potentially very different answers (figure 2).

Best-matching with cosine similarity. This constitutes finding a point $p \in S$ for a query q such that

$$\begin{aligned} p &= \arg\max_{r\in S}\frac{\langle q,r\rangle}{\|q\|\,\|r\|} = \arg\max_{r\in S}\frac{\langle q,r\rangle}{\|r\|} \\ &\neq \arg\max_{r\in S}\langle q,r\rangle \text{ (unless } \|r\| = k\ \forall\ r\in S). \end{aligned}$$

The best match with cosine similarity gives the maximum inner-product only if all the points in the set S are normalized to the same length (counter example in figure 2).

Locality-sensitive hashing. LSH has been applied to a wide variety of similarity functions. LSH involves constructing hashing functions such that each hash function h satisfies the following for any pair of points $r, p \in S$:

$$\Pr[h(r) = h(p)] = \text{sim}(r,p), \tag{2}$$

where $\text{sim}(r,p) \in [0,1]$ is the similarity function of interest. For our situation, we can normalize our data set such that $\forall\ r \in S, \|r\| \leq 1$[3], and assume that the all the data is in the first quadrant (so that none of the inner-products go below zero). In that case, $\text{sim}(r,p) = \langle r,p\rangle \in [0,1]$ is a valid similarity function of interest.

It is known that for any similarity function to admit a locality sensitive hash function family (as defined in equation 2), the distance function $\mathbf{d}(r,p) = 1 - \text{sim}(r,p)$ must satisfy the triangle inequality (Lemma 1 in [7]). However, the distance function $\mathbf{d}(r,p) = 1 - \langle r,p\rangle$ does not satisfy the triangle inequality (even when all the points are restricted to the first quadrant)[4]. So LSH cannot be applied to the inner product similarity function even when all the data lies in the first quadrant (which is quite a restrictive condition).

Efficient max-kernel operation. Various techniques have been proposed to solve this problem efficiently. For kernel functions with very high (possibly infinite) dimensional explicit representations, Rahimi, et.al., 2007 [30], propose a technique to transform these high-dimensional representa-

[3]This normalization is different than the normalization mentioned earlier where all the points were normalized to have the same length. Here the lengths are normalized to be less than equal to one, but not equal to each other.

[4]Counter example: Let $x, y, z \in S$ be points such that $\|x\| = \|y\| = \|z\| = 1$, and angles made between x & y, y & z and z & x at the origin be $\left(\frac{\pi}{4} - 0.1\right)$, $\frac{\pi}{4}$ and $\left(\frac{\pi}{2} - 0.3\right)$ respectively. The triangle inequality, $\mathbf{d}(x,y) + \mathbf{d}(y,z) \geq \mathbf{d}(z,x)$, does not hold for $d(\cdot,\cdot) = 1 - \langle\cdot,\cdot\rangle$. $\mathbf{d}(x,y) = 0.23$, $\mathbf{d}(y,z) = 0.29$ & $\mathbf{d}(z,x) = 0.70$.

tions into lower-dimensions while still approximately preserving the inner-product to improve scalability. However, the final search still involves a linear scan over the set of points for the maximum inner-product or a fast nearest-neighbor search under the assumption that finding the nearest-neighbor is equivalent to maximizing the inner-product. For translation invariant kernels[5], a tree-based recursive algorithm has been shown to scale to large sets [20]. However, it is not clear how this algorithm can be extended to the general class of kernels. LSH is widely used for image matching in computer vision [23], but only for kernel functions that admit a locality sensitive hashing function [7]. Hence, none of the existing techniques can be directly applied to maximum inner-product search without introducing inaccurate results or limiting assumptions.

2.2 Why is maximum inner-product search possibly harder?

Inner-products lack a very basic property of generally used similarity functions – coincidence. For example, the Euclidean distance of a point to itself is 0; the cosine similarity of a point to itself is 1. The inner-product of a point $x \in S$ to itself is $\|x\|^2$, which may be high or low depending on the value of the $\|x\|$. Additionally, there can possibly be many other points $y \in S$ such that $\langle y, x \rangle > \|x\|^2$.

Efficient nearest neighbor search methods typically rely heavily on these properties (triangle inequality and coincidence) to achieve their efficiency. Hence, without any added assumptions, this problem of maximum inner-product search is inherently harder than the previously dealt similar problems. This is possibly the reason why there is no existing work for this problem in its general form, to our knowledge.

2.3 Why try trees?

Trees have been widely used for nearest-neighbor search [13, 4, 29, 8, 32]. Being widely used approach in the nearest neighbor case, we believe it is instructive to review them before considering the maximum inner-product search case.

For exact nearest-neighbor searches, trees can yield great accelerations in anywhere from low- to high-dimensional data, as long as there is low intrinsic dimensionality [4, 10]. Trees can also be easily adapted to the approximate case, with error guarantees of various sorts. These include approximation in the sense of *rank*, i.e. if the actual best match may not be returned, trees can be used in a way that guarantees that the result is, say, in the top 10 best matches [32] – rather than provide a guarantee in terms of potentially less meaningful abstract quantities such as distances (as is provided by LSH; it is not clear how to extend LSH to provide rank guarantees). This would appear to make particular sense in many applications such as recommendations. Trees can also be used for another kind of approximate search setting which can be important in practical applications, in which the best possible match is found given a user-defined time limit. This kind of approximation is possible for tree-based branch-and-bound algorithms because they are incremental algorithms. This is not possible with something like LSH – LSH provides theoretical error bounds, but there is no way of ensuring the error constraint *during* the search. Another important advantage of trees is that the trees require a single

[5]Kernel functions $\mathcal{K}(p, q)$ which are dependent only the (Euclidean) distance between the points p and q are considered translation invariant kernels. The Gaussian RBF kernel is such a translation invariant kernel function.

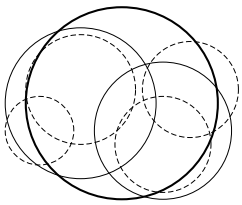

Figure 2: Ball trees

Algorithm 1 MakeBallTreeSplit(Data S)

Pick a random point $\mathbf{x} \in S$
$A \leftarrow \arg\max_{\mathbf{x}' \in S} \|\mathbf{x} - \mathbf{x}'\|_2^2$
$B \leftarrow \arg\max_{\mathbf{x}' \in S} \|A - \mathbf{x}'\|_2^2$
return (A, B)

Algorithm 2 MakeBallTree(Set of items S)

Input – Set S
Output – Tree T
$T.S \leftarrow S$ *//The set of points in node T*
$T.\mu \leftarrow \text{mean}(S)$ *//The center of the ball around T.S*
$T.R \leftarrow \max_{p \in S} \|p - T.\mu\|_2^2$ *//The radius of the ball around T.S*
if $|S| \leq N_0$ **then**
  return T
else
  $(A, B) \leftarrow$ MakeBallTreeSplit(S)
  $S_l \leftarrow \{p \in S\colon \|p - A\|_2^2 \leq \|p - B\|_2^2\}$; $S_r \leftarrow S \setminus S_l$
  $T.\text{lc} \leftarrow$ MakeBallTree(S_l) *//Left child*
  $T.\text{rc} \leftarrow$ MakeBallTree(S_r) *//Right child*
  return T
end if

Figure 3: Ball tree Construction

construction – the branch-and-bound algorithm adapts for the different levels of approximate and/or time limitations. Hashing techniques require multiple hashes for different levels of approximation. The usual norm is to pre-hash for multiple values of approximation. Trees can also be constructed by learning from the data using techniques from machine learning [6, 25] to provide better accuracy and efficiency.

The significant advantages of the tree-based approach for the nearest-neighbor setting motivate the question of whether they can be brought to the maximum inner-product case.

3. TREE-BASED SEARCH

Ball trees [29, 28] are binary space-partitioning trees that have been widely used for the task of indexing data sets. Every node in the tree represents a set of points and each node is indexed with a center and a ball enclosing all the points in the node. The set of point at a node is divided into two disjoint sets which form the child nodes, partitioning the space into (possibly overlapping) hyper-spheres. The tree is built hierarchically and a node is made a leaf if it contains a set of points of size below a threshold value N_0.

3.1 Tree construction

We use a simple ball tree construction heuristic that approximately picks a pair of pivot points which are farthest apart from each other [28], and splits the data by assigning the points to their closest pivot. The intuition behind this heuristic is that these two points might lie in the principal direction. The splitting and the recursive tree construction algorithm is presented in Algorithms 1 & 2 for completeness.

3.2 Branch-and-bound algorithm

Ball trees are widely used for the task of nearest neighbor search and are known to be fairly scalable to moderately high dimensions [28, 26]. The search usually employs

Algorithm 3 LinearSearch(Query q, Reference Set S)

```
for each p ∈ S do
  if ⟨q, p⟩ > q.λ then
    q.bm ← p
    q.λ ← ⟨q, p⟩
  end if
end for
```

Algorithm 4 TreeSearch(Query q, Tree Node T)

```
if q.λ < MIP(q, T) then
  if isLeaf(T) then
    LinearSearch(q, T.S)
  else
    I_l ← MIP(q, T.lc); I_r ← MIP(q, T.rc);
    if I_l ≤ I_r then
      TreeSearch(q, T.rc); TreeSearch(q, T.lc);
    else
      TreeSearch(q, T.lc); TreeSearch(q, T.rc);
    end if
  end if
end if
```

Algorithm 5 ExactMIP(Query set V, Reference Set S)

```
T ← MakeBallTree(S)
for each q ∈ V do
  q.bm ← ∅;   //The current max-inner-product candidate
  q.λ ← −∞; //The current highest inner-product
  TreeSearch(q, T);
  return q.bm;
end for
```

Figure 4: Single-tree Search: See text for details

the depth-first branch-and-bound algorithm – a query is answered by traversing the tree in a depth-first manner by first going down the node closer to the query and bounding the minimum possible distance to the other branch with the triangle-inequality. If this bound is greater than the distance to the current neighbor candidate for the query, the branch is removed from computation.

An analogous greedy depth-first algorithm can be used for maximum inner-product search. But instead of traversing down the node closer to the query, the choice is made on the basis of the maximum possible inner-product between the query and any potential point from the node. The recursive depth-first branch and bound algorithm is presented in Algorithm 4. The search algorithm for a query (q) begins at the root of the tree (Alg. 5). At each step, the algorithm is at a tree node (T). It checks if the maximum possible inner-product between the query and any point in the node, $\mathbf{MIP}(q, T)$, is any better than the current best-match for the query (q.bm). If the check fails, this branch of the tree is not explored any more. Otherwise, the algorithm recursively traverses the tree, exploring the branch with the better potential candidates in a depth-first manner. If the node is a leaf, the algorithm just finds the best-match within the leaf with a linear search (Alg. 3). This algorithm ensures that the exact solution (i.e., the maximum inner-product) is returned by the end of the algorithm.

3.2.1 Bounding maximum inner-product with a ball

We present an novel analytical upper bound for the maximum possible inner product of a given point (in this case, the query q) with points in a ball. It is important to note that the information about the ball is limited to its center and its radius. For the rest of this section, we use the notation $\|\cdot\|$ to denote the $\|\cdot\|_2$.

THEOREM 3.1. *Given a ball $\mathcal{B}_{p_0}^{R_p}$ of points centered at p_0 with radius R_p and (query) point q, the maximum possible inner product between the point q and the ball $\mathcal{B}_{p_0}^{R_p}$ is bounded from above by:*

$$\max_{p \in \mathcal{B}_{p_0}^{R_p}} \langle q, p \rangle \leq \langle q, p_0 \rangle + R_p \|q\| . \tag{3}$$

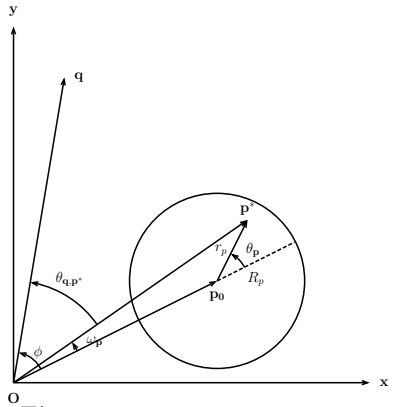

Figure 5: Bounding with a ball

PROOF. Suppose that p^* is the best possible match in the ball $\mathcal{B}_{p_0}^{R_p}$ for the query q and r_p be the Euclidean distance between the ball center p_0 and p^* (by definition, $r_p \leq R_p$). Let θ_p be the angle between the vector $\vec{p_0}$ and the vector $\vec{p_0 p^*}$, ϕ and ω_p be the angles made at the origin between the vector $\vec{p_0}$ and vectors $\vec{q}$ and $\vec{p^*}$ respectively (see figure 5). The length of p^* in terms of p_0 and θ_p is:

$$\|p^*\| = \sqrt{(\|p_0\| + r_p \cos\theta_p)^2 + (r_p \sin\theta_p)^2}. \tag{4}$$

The angle ω_p can be expressed in terms of p_0 and θ_p as:

$$\cos\omega_p = \frac{\|p_0\| + r_p \cos\theta_p}{\|p^*\|}, \sin\omega_p = \frac{r_p \sin\theta_p}{\|p^*\|}. \tag{5}$$

Let θ_{q,p^*} be the angle between the vectors $\vec{q}$ and $\vec{p^*}$. With the triangle inequality of angles, we have:

$$|\theta_{q,p^*}| \geq |\phi - \omega_p|.$$

Assuming that the angles lie in the range $[-\pi, \pi]$ (instead of the usual $[0, 2\pi]$), we get:

$$\cos\theta_{q,p^*} \leq \cos(\phi - \omega_p). \tag{6}$$

Using this inequality we obtain the following bound for the highest possible inner-product between q and any $p \in \mathcal{B}_{p_0}^{R_p}$:

$$\max_{p \in \mathcal{B}_{p_0}^{R_p}} \langle q, p \rangle = \langle q, p^* \rangle \text{(by assumption)} = \|q\| \|p^*\| \cos\theta_{q,p^*}$$

By equations 4, 5 & 6, we have

$$\begin{aligned}
\max_{p \in \mathcal{B}_{p_0}^{R_p}} \langle q, p \rangle &\leq \|q\| \|p^*\| \cos(\phi - \omega_p) \\
&= \|q\| (\cos\phi(\|p_0\| + r_p \cos\theta_p) + \sin\phi(r_p \sin\theta_p)) \\
&\leq \|q\| \max_{\theta_p} (\cos\phi(\|p_0\| + r_p \cos\theta_p) + \sin\phi(r_p \sin\theta_p)) \\
&= ||q|| (\cos\phi(||p_0|| + r_p \cos\phi) + \sin\phi(r_p \sin\phi)) \\
&\leq \|q\| (\cos\phi(\|p_0\| + R_p \cos\phi) + \sin\phi(R_p \sin\phi)) .
\end{aligned}$$

The third inequality comes from the definition of maximum. The following equality comes from maximizing over θ_p. This gives us the optimal value of $\theta_p = \phi$. The final inequality is comes from the fact that $r_p \leq R_p$. Simplifying the final inequality gives us equation 3. □

For the tree-search algorithm (Alg. 4), we set the maximum possible inner-product between q and a tree node T as

$$\mathbf{MIP}(q, T) = \langle q, T.\mu \rangle + T.R \|q\| .$$

This upper bound can be computed in almost the same time required for a single inner-product (since the norms of the queries can be pre-computed before searching the tree).

4. DUAL-TREE BASED SEARCH

For a set of queries, the tree can be traversed separately for each query. However, if the set of queries is very large, a common technique to improve efficiency of querying is to index the queries in the form of a tree as well. The search is performed by traversing both trees simultaneously using the *dual-tree* algorithm [15]. The basic idea is to amortize the cost of tree-traversal for a set of similar queries. The dual-tree algorithms have been applied to different tree-based algorithms like nearest-neighbor search [15] and kernel density estimation [16] with theoretical runtime guarantees [31].

4.1 Dual-tree branch-and-bound algorithm

The generic dual-tree algorithm is presented in Alg. 6. Similar to the Alg. 4, the algorithm traverses down the tree on the reference set S (*RTree*). However, the algorithm also traverses down the tree on the set V of queries (*QTree*), resulting in a four-way recursion. At each step, the algorithm is at a QTree node Q and a RTree node T. For every Q, the value $Q.\lambda$ denotes the minimum inner-product between any query in Q and its current best-match candidate. If this value is greater than the maximum possible inner product, $\mathbf{MIP}(Q, T)$, between any query in Q and any reference point in T, this part of the recursion is no longer explored. When the algorithm is at the leaf level of both the trees, it obtains the best-matches for each query in the QTree leaf by doing a linear scan over the RTree leaf.

We explore two ways of indexing the queries – (1) indexing the queries using the ball-tree (*MakeQueryTree* in Alg.7 is Alg. 2) (2) indexing the queries using a novel data structure, the *cone-tree* (*MakeQueryTree* in Alg.7 is Alg. 9). In the following subsection, we derive expressions for $\mathbf{MIP}(Q, T)$ for the ball-tree. The expressions for the cone-tree is presented in section 5.

4.2 Using ball trees

In this subsection, we provide inner-product bounds between two balls with the following theorem:

THEOREM 4.1. *Given two balls $\mathcal{B}_{p_0}^{R_p}$ and $\mathcal{B}_{q_0}^{R_q}$ centered at p_0 and q_0 with radius R_p and R_q respectively, the maximum possible inner-product with any pair of points $p \in \mathcal{B}_{p_0}^{R_p}$ and $q \in \mathcal{B}_{q_0}^{R_q}$ is bounded from above by:*

$$\langle q_0, p_0 \rangle + R_q R_p + \|q_0\| R_p + \|p_0\| R_q. \quad (7)$$

PROOF. Consider the pair of point $(p^*, q^*), p^* \in B_{p_0}^{R_p}, q^* \in B_{q_0}^{R_q}$ be such that

$$\langle q^*, p^* \rangle = \max_{p \in B_{p_0}^{R_p}, q \in B_{q_0}^{R_q}} \langle q, p \rangle. \quad (8)$$

Let θ_p be the angle $\vec{p_0}$ makes with the vector $p_0\vec{p}^*$, and θ_q be the corresponding angle in the query ball. Let ω_p be the angle between the vectors $\vec{p_0}$ and $\vec{p^*}$ and ω_q be the angle between the vectors $\vec{q_0}$ and $\vec{q^*}$. Let r_p be the distance between p_0 and p^*, r_q be the distance between q_0 and q^*. Let ϕ be the angle made between p_0 and q_0 at the origin.

Some facts for the ball $\mathcal{B}_{p_0}^{R_p}$ (the facts are analogous for the ball $\mathcal{B}_{q_0}^{R_q}$):

$$\|p^*\| = \sqrt{\|p_0\|^2 + r_p^2 + 2\|p_0\| r_p \cos\theta_p},$$

$$\cos\omega_p = \frac{\|p_0\| + r_p \cos\theta_p}{\|p^*\|}, \sin\omega_p = \frac{r_p \sin\theta_p}{\|p^*\|}.$$

Using the triangle inequality of the angles, we know that:

$$|\theta_{q^*,p^*}| \geq |\phi - (\omega_p + \omega_q)|,$$

giving us the following:

$$\langle q^*, p^* \rangle = \|p^*\| \|q^*\| \cos(\phi - (\omega_p + \omega_q)). \quad (9)$$

Algorithm 6 DualSearch(QTree Node Q, RTree Node T)

```
if Q.λ < MIP(Q, T) then
  if isLeaf(T) & isLeaf(Q) then
    for each q ∈ Q.S do
      LinearSearch(q, T.S)
    end for
    Q.λ ← min_{q∈Q.S} q.λ
  else if isLeaf(T) then
    DualSearch(Q.lc, T); DualSearch(Q.rc, T);
    Q.λ ← min{Q.lc.λ, Q.rc.λ}
  else if isLeaf(Q) then
    I_l ← MIP(Q, T.lc); I_r ← MIP(Q, T.rc);
    if I_l ≤ I_r then
      DualSearch(Q, T.rc); DualSearch(Q, T.lc);
    else
      DualSearch(Q, T.lc); DualSearch(Q, T.rc);
    end if
  else
    I_l ← MIP(Q.lc, T.lc); I_r ← MIP(Q.lc, T.rc);
    if I_l ≤ I_r then
      DualSearch(Q.lc, T.rc); DualSearch(Q.lc, T.lc);
    else
      DualSearch(Q.lc, T.lc); DualSearch(Q.lc, T.rc);
    end if
    I_l ← MIP(Q.rc, T.lc); I_r ← MIP(Q.rc, T.rc);
    if I_l ≤ I_r then
      DualSearch(Q.rc, T.rc); DualSearch(Q.rc, T.lc);
    else
      DualSearch(Q.rc, T.lc); DualSearch(Q.rc, T.rc);
    end if
    Q.λ ← min{Q.lc.λ, Q.rc.λ}
  end if
end if
```

Algorithm 7 ExactMIPDT(Query Set V, Reference Set S)

```
T ← MakeBallTree(S)
Q ← MakeQueryTree(V)
∀ trees nodes Q' in the tree Q, Q'.λ ← −∞;
∀ queries q ∈ V, q.bm ← ∅, q.λ ← −∞;
DualSearch(Q, T);
∀ queries q ∈ V, return q.bm;
```

Figure 6: Dual-tree Search: See text for details.

Replacing ω_p and ω_q with θ_p and θ_q by using the aforementioned equalities (similar to the techniques in proof for theorem 3.1), we have:

$$\begin{aligned}
\langle q^*, p^* \rangle &= \langle q_0, p_0 \rangle + r_p r_q \cos(\phi - (\theta_p + \theta_q)) \\
&\quad + r_p \|q_0\| \cos(\phi - \theta_p) + r_q \|p_0\| \cos(\phi - \theta_q) \\
&\leq \max_{\theta_p, \theta_q} \langle q_0, p_0 \rangle + r_p r_q \cos(\phi - (\theta_p + \theta_q)) \\
&\quad + r_p \|q_0\| \cos(\phi - \theta_p) + r_q \|p_0\| \cos(\phi - \theta_q) \\
&\leq \max_{r_p, r_q} \langle q_0, p_0 \rangle + r_p r_q + r_q \|p_0\| + r_p \|q_0\| \\
&\leq \langle q_0, p_0 \rangle + R_p R_q + R_q \|p_0\| + R_p \|q_0\|,
\end{aligned}$$

where the first inequality comes from the definition of max. The second inequality follows from $\cos(\cdot) \leq 1$ and the final inequality comes from the fact that $r_p \leq R_p$, $r_q \leq R_q$. □

For the dual-tree search algorithm (Alg. 6), the maximum-possible inner-product between two tree nodes Q and T is:

$$\mathbf{MIP}(Q, T) = \langle q_0, p_0 \rangle + R_p R_q + R_q \|p_0\| + R_p \|q_0\|.$$

It is interesting to note that this upper bound bound reduces to the bound in theorem 3.1 when the ball containing the queries is reduced to a single point, implying $R_q = 0$.

5. CONE TREES

In equation 1, the point p, where the maximum is achieved, is independent of the norm $||q||$ of the query q. Let $\theta_{q,r}$ be the angle between the q and r at the origin, then the task

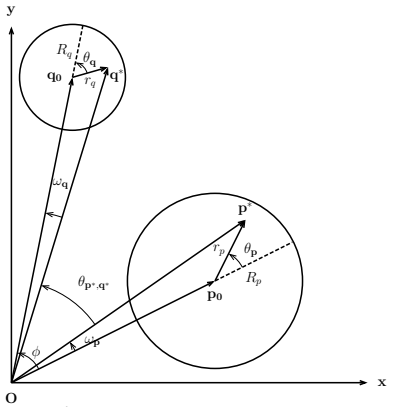

Figure 7: Bounding between two balls.

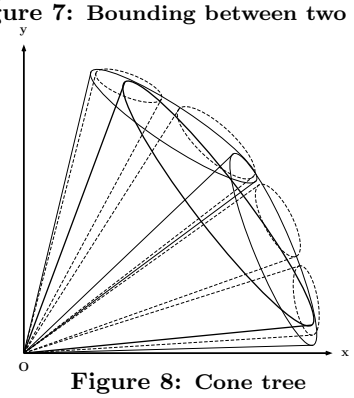

Figure 8: Cone tree

of maximum inner-product search is equivalent to finding a point $p \in S$ such that:

$$p = \arg\max_{r \in S} \|r\| \cos\theta_{q,r}. \quad (10)$$

This implies that only the directions of the queries affect the solution. Balls provide bounds on the inner-product since they bound the norm of the vector as well as the direction. Since the norms do not matter for the queries, indexing them in balls is not required (and hence bounding their norms) is not necessary. Only the range of their directions need to be bounded. For this reason, we propose the indexing of the queries on the basis of their direction (from the origin) to form a *cone tree* (figure 8). The queries are hierarchically indexed as (possibly overlapping) open cones. Each cone is represented by a vector, which corresponds to its axis, and an angle, which corresponds to its aperture[6].

5.1 Cone tree construction

The cone tree construction is very similar to the ball tree construction. The only difference is the use of cosine similarity instead of the Euclidean distances for the task of splitting (pseudo-code in Figure 8).

5.2 Cone-ball bound

Since the norms of the queries do not affect the solution in equation 10, we assume that all the queries have unit norm.

THEOREM 5.1. *Given a ball $\mathcal{B}_{p_0}^{R_p}$ of points centered at p_0 with radius R_p and a cone $\mathcal{C}_{q_0}^{\omega_q}$ of queries (normalized to length 1) with the axis of the cone q_0 and aperture of $2\omega_q \geq 0$, the maximum possible inner-product between any pair of points $p \in \mathcal{B}_{p_0}^{R_p}$, $q \in \mathcal{C}_{q_0}^{\omega_q}$ is bounded from above by:*

$$\|p_0\| \cos(\{|\phi| - \omega_q\}_+) + R_p, \quad (11)$$

where ϕ is the angle made between p_0 and q_0 at the origin and the function $\{x\}_+ = \max\{x, 0\}$.

[6]The aperture of the cone is twice the angle made between the axis and the perimeter of the cone.

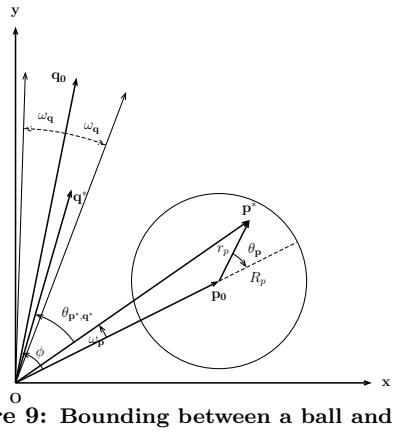

Figure 9: Bounding between a ball and a cone

Algorithm 8 MakeConeTreeSplit(Data Q)

Pick a random point $\mathbf{x} \in Q$
$A \leftarrow \arg\min_{\mathbf{x}' \in S} \cos\theta_{\mathbf{x},\mathbf{x}'}$
$B \leftarrow \arg\min_{\mathbf{x}' \in S} \cos\theta_{A,\mathbf{x}'}$
return (A, B).

Algorithm 9 MakeConeTree(Set of items S)

Input – Set S
Output – Tree T
$T.S \leftarrow S$ *//The set of points in node T*
$T.\mu \leftarrow \text{mean}(S)$ *//The axis of the cone around T.S*
$T.C \leftarrow \min_{p \in S} \cos\theta_{T.\mu,p}$ *//The cosine of the aperture of the cone*
if $|S| \leq N_0$ **then**
  return T
else
  $(A, B) \leftarrow$ MakeConeTreeSplit(S)
  $S_l \leftarrow \{p \in S : \cos\theta_{A,p} > \cos\theta_{B,p}\}$; $S_r \leftarrow S \setminus S_l$
  T.lc $\leftarrow$ MakeConeTree(S_l) *//Left child*
  T.rc $\leftarrow$ MakeConeTree(S_r) *//Right child*
  return T
end if

Figure 10: Cone tree Construction

PROOF. There are two cases to consider here:
(i) $|\phi| < \omega_q$
(ii) $|\phi| \geq \omega_q$

For case (i), the center p_0 of the ball $\mathcal{B}_{p_0}^{R_p}$ lies within the cone $\mathcal{C}_{q_0}^{\omega_q}$, implying that

$$\max_{q \in \mathcal{C}_{q_0}^{\omega_q}, p \in \mathcal{B}_{p_0}^{R_p}} \|p\| \cos\theta_{q,p} \leq \|p_0\| + R_p. \quad (12)$$

since there could be some query $q^* \in \mathcal{C}_{q_0}^{\omega_q}$ which is in the same direction as p_0, giving the maximum possible inner-product.

For case (ii), let us assume that $\phi \geq 0$ without loss of generality. Then $\phi \geq \omega_q$. Continuing with the similar notation as in theorem 3.1 & 4.1 for the best pair of points (q^*, p^*) as well as the notation from figure 9, we can say that

$$|\theta_{p^*,q^*}| \geq |\phi - \omega_q - \omega_p| \quad (13)$$

Since ω_q is fixed, we can say that

$$\max_{q \in C_{q_0}^{\omega_q}, p \in B_{p_0}^{R_p}} \|p\| \cos\theta_{q,p} \leq \|p^*\| \cos\theta_{q^*,p^*} \text{ (by def.)}$$

$$\leq \|p^*\| \cos(\phi - \omega_q - \omega_p). \quad (14)$$

Expressing $\|p^*\|$ and ω_p in terms of $\|p_0\|, r_p$ and θ_p, and then subsequently maximizing over θ_p and using the fact that $r_p \leq R_p$, we get that

$$\max_{q \in C_{q_0}^{\omega_q}, p \in B_{p_0}^{R_p}} \|p\| \cos\theta_{q,p} \leq \|p_0\| \cos(\phi - \omega_q) + R_p. \quad (15)$$

Combining case (i) and (ii), we obtain equation 11. □

Dataset	Dimensions	Reference set	Query Set
Bio	74	210,409	75,000
Corel	32	27,749	10,000
Covertype	55	431,012	150,000
LCDM	3	10,777,216	6,000,000
LiveJournal	25,327	121,625	100,000
MNIST	786	60,000	10,000
MovieLens	51	3,706	6,040
Netflix	51	17,770	480,189
OptDigits	64	1,347	450
Pall7	7	100,841	100,841
Physics	78	112,500	37,500
PSF	2	3,056,092	3,056,092
SJ2	2	50,000	50,000
U-Random	20	700,000	300,000
Y!-Music	51	624,961	1,000,990

Table 1: Datasets used for evaluation

6. EXPERIMENTS AND RESULTS

In this section, we evaluate the efficiency of algorithms 5(SB – single ball tree) & 7. For the dual-tree algorithm, we use the two variations – (i) the set of queries indexed as a ball tree (DBB – dual ball-ball), (ii) the set of queries indexed as a cone tree (DBC – dual ball-cone). We compare our proposed algorithms to the linear search presented in Alg. 3 (LS – linear search). We report the speedup[7] of the proposed algorithms over linear search. For the trees, the leaf size N_0 can be selected by cross-validation. However, for our experiments, we choose a ad hoc value of $N_0 = 20$ for all datasets to demonstrate the gain in efficiency without any expensive cross-validation.

Datasets. We use a variety of datasets from different fields of data mining. We use the following collaborative filtering datasets: MovieLens [17], Netflix [3] and the Yahoo! Music [12] datasets. For text data, we use the LiveJournal blog moods data set [19]. We also use the MNIST digits dataset [24] for evaluation. Three astronomy datasets, LCDM [27], PSF and SJ2, are also considered. A synthetic data set (U-Rand) of uniformly random points in 20 dimensions is used. The rest of the datasets are widely used machine learning data sets from the UCI machine learning repository [5]. The details of the dataset sizes are presented in Table 1[8].

Tree construction times. The tree-building procedure is extremely efficient. We present the tree construction times in table 6 and contrast them with the runtime of the linear search algorithm. In the last column, we present the ratio of the tree construction times with the runtimes of Alg. 3. For the single ball and dual ball-ball algorithm, the tree construction involves building one and two ball trees respectively. For the dual ball-cone algorithm, the queries are normalized to have unit length for convenience[9]. Following the query normalization, two trees are built. The time required for query normalization is included in the construction time. This accounts for the significant difference between construction times for the dual ball-ball and the dual ball-cone algorithm.

The numbers in the last column of table 6 (R) show how small the construction times are with respect to the actual linear search. The highest ratio is 0.15 for the OptDigits dataset. This implies that any speedup over 1.18 at search time is enough to compensate for the tree construction time. For most of the datasets, this ratio is much lower. Moreover, this tree building cost is a one time cost. Once the tree is built, it can be used for searching the dataset multiple times.

[7]Speedup is defined as the ratio of the time taken by the linear search and the time taken by the evaluated algorithm.

[8]For the collaborative filtering datasets, the references (the items) and the queries (the users) are clearly defined. We randomly split the datasets into queries (V) and references (S) for the rest.

[9]This is because the query norms do not affect the answers.

Dataset	SB	DBB	DBC	LS	R(%)
Bio	4.3	5.7	10.6	4,028	0.25
Corel	0.2	0.27	0.66	43	1.5
Covertype	5.5	7.2	14.8	14,885	0.1
LCDM	36.7	56.46	99.3	1,984,200	0.005
LiveJournal	2223	4073	4745	517,194	0.92
MNIST	8.06	9.1	11.38	817	1.5
MovieLens	0.03	0.08	0.27	4.62	6
Netflix	0.2	8.27	33.5	1,878	1.7
OptDigits	0.01	0.012	0.022	0.135	15
Pall7	0.26	0.52	1.4	364	0.4
Physics	2.33	3.0	5.8	1,114	0.5
PSF	9.06	18.1	34.95	282,514	0.01
SJ2	0.1	0.2	0.46	75	0.6
U-Rand	4.94	6.9	15.64	26,586	0.6
Y! Music	9.72	28.85	112.5	137,306	0.08

Table 2: Tree construction time (in seconds) contrasted with the linear search time (in seconds).

Dataset	SB	DBB	DBC
Bio	7,059.62	6.55	273.52
Corel	14.27	17.38	7.68
Covertype	927.51	10.05	773.34
LCDM	29,526	1,327	101,950
LiveJournal	8.36	1.59	15.45
MNIST	2.61	2.22	2.5
MovieLens	2.23	1.36	1.67
Netflix	1.98	1.92	1.84
OptDigits	1.13	1.10	1.10
Pall7	1,020	23.14	2,285
Physics	4.93	4.0	4.08
PSF	61,502	96,570	125,800
SJ2	544	190	767
U-Rand	3.76	3.18	3.28
Y!-Music	2.11	2.09	2.16

Table 3: Speedups over linear search for $k = 1$

Search efficiency. The speedups over linear search are presented in Table 6. Overall, the speedup numbers vary from as low as 1.13 for the OptDigits dataset to over 10^5 (4 orders of magnitude) for the LCDM and the PSF dataset. An important thing to note here is that for datasets with low speedup (below an order of magnitude) with the single ball tree algorithm, the speedup numbers for all three algorithms were pretty low and fairly comparable. However, even a speedup of 2 is pretty significant in terms of absolute times. For example, for the Yahoo! music dataset, a search speedup of mere 2 with a tree construction time of 120 seconds gives a saving of 19 hours of computation time. For most datasets with a high value of speedup for single ball tree algorithm, the speedups for the dual-tree algorithms are also very high.

There are three important things to note here. Firstly, the dual-tree algorithms (Alg. 7) do not perform very well if the single-tree algorithms (Alg. 5) does not have a high speedup. This is mostly because the tree is unable to find tight bounds and hence has to travel every branch. The dual-tree scheme loosens the bound to amortize the traversal cost over multiple queries. But if the bounds are bad for algorithm 5, the bounds for the dual-tree are much worse. Hence, the dual-tree algorithm does not show any significant speedup. Secondly, the dual-tree algorithm (especially dual ball-cone) starts outperforming the single-tree algorithm sig-

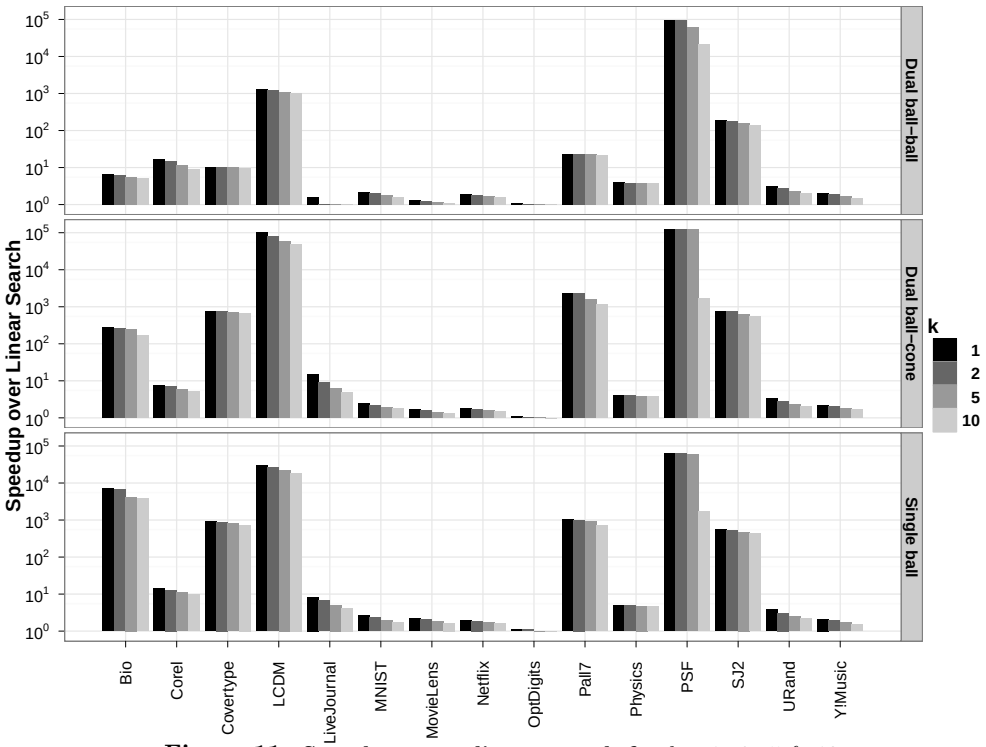

Figure 11: Speedups over linear search for $k = 1,\ 2,\ 5$ & 10.

nificantly when the set of queries is really large. This is a usual behavior for dual-tree algorithms. The query set has to be large enough for the gains from the amortization of query traversal of the reference tree (*RTree*) to outweigh the computational cost of traversing the query-tree (*QTree*) itself. Finally, the dual-tree algorithm with ball trees for the query set is generally significantly slower than the dual-tree with a cone tree for the queries. There are possibly two possible reasons for that – (i) The cones provide a tighter indexing of the queries than balls. A single cone can be used to index points in multiple balls which lie in the same direction but have varying norms. (ii) The upper bound for $\mathbf{MIP}(Q, T)$ in equation 10 is fairly loose.

We also consider the general problem of obtaining the points in the set S with the k highest inner-product with the query q. This is analogous to the k-nearest neighbor search problem. We present the speedups of our algorithms over linear search for $k = 1,\ 2,\ 5$ & 10 in figure 11.

7. MAX-KERNEL OPERATION WITH GENERAL KERNEL FUNCTIONS

In this section, we provide some discussion of how the proposed algorithms might be applied in an inner-product space without the explicit representation of the points in the inner-product space. The inner-products are defined by a kernel function $\mathcal{K}(q, p) = \langle \varphi(q), \varphi(p) \rangle$.

The tree construction has to be modified to work in the inner-product space. For a tree node T with the set of point $T.S$, the mean in φ-space is defined as $\mu = \frac{1}{|T.S|} \sum_{p \in T.S} \varphi(p)$. However, μ might not have an explicit representation, but it is possible to compute inner products with μ as follows:

$$\langle \mu, \varphi(q) \rangle = \frac{1}{|T.S|} \sum_{p \in T.S} \mathcal{K}(q, p).$$

However, this computation is possibly very expensive during search time. Hence we propose picking the point in the φ-space which is closest to the mean μ as the new center. So the new center p_c is given by:

$$p_c = \arg \min_{r \in T.S} \mathcal{K}(r, r) - \frac{2}{|T.S|} \sum_{r' \in T.S} \mathcal{K}(r', r). \quad (16)$$

This operation is quadratic in computation time, but is done at the preprocessing phase to provide efficiency during the search phase. Given this new center p_c, the radius R_p of the ball enclosing the set $T.S$ in φ-space is given by:

$$R_p^2 = \max_{r \in T.S} \mathcal{K}(p_c, p_c) + \mathcal{K}(r, r) - 2\mathcal{K}(r, p_c). \quad (17)$$

With these definitions for the center and the radius, a ball tree can be built in any φ-space using Algorithm 2. Given a ball in φ-space, the equation 3 in theorem 3.1 becomes:

$$\mathbf{MIP}(q, T) = \mathcal{K}(q, p_c) + R_p \sqrt{\mathcal{K}(q, q)}. \quad (18)$$

Computing this upper bound is equivalent to a single kernel function evaluation ($\mathcal{K}(q, q)$ be pre-computed before searching the tree). Using this upper bound, the tree-search algorithm (Alg. 5) can be performed in any φ-space. We will evaluate this method in the longer version of the paper.

Using the same principles, the dual-tree algorithm (Alg. 7) can also be applied to any φ-space. For the dual-tree with ball-tree for the queries, the upper bound (eq. 7) on the maximum inner-product between queries in node Q and points in node T in theorem 4.1 is modified to:

$$\mathcal{K}(q_c, p_c) + R_p R_q + R_p \sqrt{\mathcal{K}(q_c, q_c)} + R_q \sqrt{\mathcal{K}(p_c, p_c)}, \quad (19)$$

where p_c and q_c are the chosen ball centers in the φ-space with radius R_p and R_q respectively.

For queries indexed in a cone-tree, the central axis of the cone can be the point in the φ-space making the smallest angle with the mean of the set in the φ-space. Since the queries are supposed to be normalized in the φ-space, for a query tree node Q, the mean of the set $Q.S$ is supposed to be $\mu = \frac{1}{|Q.S|} \sum_{q \in Q.S} \frac{\varphi(q)}{\|\varphi(q)\|}$. So the new central axis q_c of the cone is given by:

$$q_c = \arg \max_{q \in Q.S} \frac{\sum_{q' \in Q.S} \frac{\mathcal{K}(q', q)}{\sqrt{\mathcal{K}(q', q')}}}{\sqrt{\mathcal{K}(r, r)}}. \quad (20)$$

Again, this computation is quadratic in the size of the dataset, but provides efficiency during search time. The cosine of half

the aperture of the cone is now given by:

$$\cos \omega_q = \min_{q \in Q.S} \frac{\mathcal{K}(q_c, q)}{\sqrt{\mathcal{K}(q_c, q_c)\mathcal{K}(q, q)}}. \quad (21)$$

The upper bound in theorem 5.1 for a cone-tree node Q of queries and a ball-tree node T of reference points becomes:

$$\sqrt{\mathcal{K}(p_c, p_c)} \cos(\{|\phi| - \omega_q\}_+) + R_p, \quad (22)$$

where ϕ is defined as:

$$\cos \phi = \frac{\mathcal{K}(p_c, q_c)}{\sqrt{\mathcal{K}(q_c, q_c)\mathcal{K}(p_c, p_c)}}.$$

This bound is very efficient to compute as it only requires a single kernel function evaluation (the terms $\mathcal{K}(p_c, p_c)$ and $\mathcal{K}(q_c, q_c)$ can be pre-computed and stored in the trees).

8. CONCLUSION

We consider the general problem of maximum inner-product search and present three novel methods to solve this problem efficiently. We use the tree data structure and present a branch-and-bound algorithm for maximum inner-product search. We also present a dual tree algorithm for multiple queries. We evaluate the proposed algorithms with a variety of datasets and exhibit their computational efficiency.

A theoretical analyses of these proposed algorithms would give us a better understanding of the computational efficiency of these algorithms. A rigorous analysis of the runtime for our algorithm would be part of our future work.

9. REFERENCES

[1] R. Bayardo, Y. Ma, and R. Srikant. Scaling Up All Pairs Similarity Search. In *Proceedings of the 16th Intl. Conf. on World Wide Web*, 2007.

[2] R. M. Bell and Y. Koren. Lessons from the Netflix Prize Challenge. *SIGKDD Explor. Newsl.*, 2007.

[3] J. Bennett and S. Lanning. The Netflix Prize. In *Proc. KDD Cup and Workshop*, 2007.

[4] A. Beygelzimer, S. Kakade, and J. Langford. Cover Trees for Nearest Neighbor. *Proceedings of the 23rd Intl. Conf. on Machine Learning*, 2006.

[5] C. L. Blake and C. J. Merz. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/, 1998.

[6] L. Cayton and S. Dasgupta. A Learning Framework for Nearest Neighbor Search. *Advances in Neural Info. Proc. Systems 20*, 2007.

[7] M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the 34th annual ACM Symp. on Theory of Comp.*, 2002.

[8] P. Ciaccia and M. Patella. PAC Nearest Neighbor Queries: Approximate and Controlled Search in High-dimensional and Metric spaces. *Proceedings of 16th Intl. Conf. on Data Engineering*, 2000.

[9] K. Clarkson. Nearest-neighbor Searching and Metric Space Dimensions. *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, 2006.

[10] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th annual ACM Symp. on Theory of Comp.*, 2008.

[11] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society of Info. Science*, 1990.

[12] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The Yahoo! Music Dataset and KDD-Cup'11. *Journal Of Machine Learning Research*, 2011.

[13] J. H. Freidman, J. L. Bentley, and R. A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.*, 1977.

[14] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. *Proceedings of the 25th Intl. Conf. on Very Large Data Bases*, 1999.

[15] A. G. Gray and A. W. Moore. 'N-Body' Problems in Statistical Learning. In *Advances in Neural Info. Proc. Systems 13*, 2000.

[16] A. G. Gray and A. W. Moore. Nonparametric Density Estimation: Toward Computational Tractability. In *SIAM Data Mining*, 2003.

[17] GroupLens. MovieLens dataset.

[18] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th annual ACM Symp. on Theory of Comp.*, 1998.

[19] S. Kim, F. Li, G. Lebanon, and I. Essa. Beyond Sentiment: The Manifold of Human Emotions. *Arxiv preprint* `arXiv:1202.1568`, 2011.

[20] M. Klaas, D. Lang, and N. de Freitas. Fast Maximum-a-posteriori Inference in Monte Carlo State Spaces. In *Artificial Intelligence and Statistics*, 2005.

[21] Y. Koren. The BellKor solution to the Netflix Grand Prize. 2009.

[22] Y. Koren, R. M. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems. *IEEE Computer*, 2009.

[23] B. Kulis and K. Grauman. Kernelized Locality-sensitive Hashing for Scalable Image Search. In *IEEE 12th Intl. Conf. on Computer Vision*, 2009.

[24] Y. LeCun. MNIST dataset, 2000. http://yann.lecun.com/exdb/mnist/.

[25] Z. Li, H. Ning, L. Cao, T. Zhang, Y. Gong, and T. S. Huang. Learning to Search Efficiently in High Dimensions. In *Advances in Neural Info. Proc. Systems 24*. 2011.

[26] T. Liu, A. W. Moore, A. G. Gray, and K. Yang. An Investigation of Practical Approximate Nearest Neighbor Algorithms. In *Advances in Neural Info. Proc. Systems 17*, 2005.

[27] R. Lupton, J. Gunn, Z. Ivezic, G. Knapp, S. Kent, and N. Yasuda. The SDSS Imaging Pipelines. *Arxiv preprint astro-ph/0101420*, 2001.

[28] S. M. Omohundro. Five Balltree Construction Algorithms. Technical report, International Computer Science Institute, December 1989.

[29] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, 1985.

[30] A. Rahimi and B. Recht. Random Features for Large-scale Kernel Machines. *Advances in Neural Info. Proc. Systems 20*, 2007.

[31] P. Ram, D. Lee, W. March, and A. Gray. Linear-time Algorithms for Pairwise Statistical Problems. In *Advances in Neural Info. Proc. Systems 22*. 2009.

[32] P. Ram, D. Lee, H. Ouyang, and A. G. Gray. Rank-Approximate Nearest Neighbor Search: Retaining Meaning and Speed in High Dimensions. In *Advances in Neural Info. Proc. Systems 22*. 2009.