Efficient Retrieval of Recommendations in a Matrix Factorization Framework

Noam Koenigstein
School of Electrical Engineering
Tel Aviv University

Parikshit Ram
Computational Science & Engineering
Georgia Institute of Technology

Yuval Shavitt
School of Electrical Engineering
Tel Aviv University

ABSTRACT

Low-rank Matrix Factorization (MF) methods provide one of the simplest and most effective approaches to collaborative filtering. This paper is the first to investigate the problem of efficient retrieval of recommendations in a MF framework. We reduce the retrieval in a MF model to an *apparently* simple task of finding the maximum dot-product for the user vector over the set of item vectors. However, the problem of efficiently finding the maximum dot-product in the general case has never been studied to the best of our knowledge. To this end, we propose two techniques for efficient search – (i) We index the item vectors in a binary spatial-partitioning metric tree and use a simple branch-and-bound algorithm with a novel bounding scheme to efficiently obtain exact solutions. (ii) We use spherical clustering to index the users on the basis of their preferences and pre-compute recommendations only for the representative user of each cluster to obtain extremely efficient approximate solutions. We obtain a theoretical error bound which determines the quality of any approximate result and use it to control the approximation. Both these simple techniques are fairly independent of each other and hence are easily combined to further improve recommendation retrieval efficiency. We evaluate our algorithms on real-world collaborative-filtering datasets, demonstrating more than ×7 speedup (with respect to the naive linear search) for the exact solution and over ×250 speedup for approximate solutions by combining both techniques.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms

Keywords

inner-product, fast retrieval, collaborative filtering

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.
Copyright 2012 ACM 978-1-4503-1156-4/12/10 ...$10.00.

1. INTRODUCTION

Recommender systems based on Matrix Factorization (MF) models have repeatedly demonstrated better accuracy than other methods such as nearest neighbor models and restricted Boltzmann machines [1, 8]. However, large scale MF models for real world recommender systems (e.g., [5, 12, 7]) run into a difficulty rarely discussed in the academic literature – the computational cost of finding the top-rated items for every user in the system once the model has been trained.

In MF models, the predicted rating of a user for an item boils down to a dot-product between two vectors representing the user and the item (we will discuss this explicitly in section 2). Constructing the entire *#USERS* × *#ITEMS* preference matrix (or even just for the top-rated items for every user) requires heavy computational (and space) resources. For example, the recently published Yahoo! Music dataset [8] has 1,000,990 users and 624,961 music items. Generating the optimal recommendations in this dataset requires over 6×10^{12} dot-products using a naive algorithm. A 50-dimensional model required 134 hours (over 5 days) to find optimal recommendations for all the users[1]. In terms of storage, saving the whole preference matrix requires over 5TB of disk-space. Moreover, this dataset of 10^6 users is just a small sample of the actual Yahoo! Music dataset and the problem worsens with larger numbers.

The problem of finding the maximum dot-product for a given query p (in this case, a user) over a set of points S (in this case, the items) is to find the point $q \in S$ such that:

$$p^\top q = \max_{q_i \in S} p^\top q_i. \tag{1}$$

Surprisingly, we did not find any technique to efficiently solve this problem; a linear search over the set of points appears to be the state-of-the-art! Moreover, the number of queries in our application is very high (possibly higher than the number of points). The focus of our paper is *to develop algorithms to solve* (1) *for multiple queries more efficiently than the linear-search algorithm.* Our contributions are:

- A simple branch-and-bound algorithm on a tree index with a novel bound to solve the exact problem.
- An approximate scheme that pre-computes solutions for certain representative queries and uses these solutions for the new queries. This makes the retrieval process extremely efficient. The representative queries are obtained by clustering the available queries.
- A theoretical error bound which determines the quality of the approximate results for a new query and is used to control the approximation by adaptively rejecting overly

[1] Using an Intel Xeon (E7320) CPU running at 2.13GHz.

(theoretically) inaccurate solutions and recomputing exact solutions for the query.

Parallelization.

The computational cost of recommendation retrieval can be mitigated by parallelization. One possible way of parallelizing involves dividing the users across cores/machines – each worker can compute the recommendations for a single user (or a small set of users). However, this is wasteful in resources and requires complex setups. Moreover, this form of parallelization does not mitigate the high latency of computing recommendations for a single user. Map-reduce parallelization can reduce the single user latency. However, a single map-reduce can at best take $O(\sqrt{\#ITEMS})$ time for the retrieval task of a single user.

Our proposed techniques are orthogonal to parallelization, and can be parallelized to improve the scalability. The branch-and-bound algorithm reduces single query latency (and can have $O(\log(\#ITEMS))$ retrieval time). Our approximate scheme reduces the number of users by only choosing a small set of representative users.

Efficient Retrieval Algorithms.

The problem of efficient Retrieval of Recommendations (RoR) in collaborative filtering has been previously studied. Large-scale recommender systems use techniques like min-hash clustering of users, probabilistic latent semantic indexing, and co-visitation counts to achieve fair scalability [5]. A more recent method based on multidimensional scaling embed both the users and items in a common Euclidean space [11], reducing the retrieval task to the problem of k-nearest-neighbor search. The plethora of algorithms for nearest-neighbor search can then be used for efficient RoR. While these methods show significant improvements in retrieval times, they deviate from the more accurate MF framework. *Our proposed methods are the first efficient retrieval algorithms in the MF framework.*

This paper.

In section 2, we introduce the MF framework and reduce the task of RoR to a relatively simpler task of finding the best-matched items with respect to the dot-product of their representative vectors with the vector representing the user. Section 3 contrasts this dot-product based best matching problem to existing best matching problems in literature. In section 4, we index the items as a Metric tree and then propose a novel branch-and-bound algorithm to efficiently obtain the exact top predicted preferences for a single user. To obtain further scalability, section 5 presents an approximated method by clustering users with “similar tastes”. The efficiency is obtained by pre-computing the top recommendations for the representative users (the cluster centers). We also present the theoretical worst-case error bound used to control the approximation. For further improvement, we combine both techniques and evaluate our proposed methods in section 6 on prominent collaborative filtering datasets.

Notation.

We reserve special indexing letters for distinguishing users from items: for users u and v, and for items i and j. A rating r_{ui} indicates the rating given by user u to item i. We denote by $\theta_{x,y}$ the angle between the vectors x and y at the origin. Finally, we denote the l_2-norm of a vector x as $\|x\|$.

2. MATRIX FACTORIZATION

In MF models, each user u is associated with a user-traits vector $p_u \in \mathbb{R}^D$, and each item i with an item-traits vector $q_i \in \mathbb{R}^D$. Predicted ratings are obtained using the rule:

$$\hat{r}_{ui} = \mu + b_i + b_u + p_u^\top q_i, \tag{2}$$

where μ is the overall mean rating value and b_i and b_u are scalers that represent the item and user biases respectively.

A user bias models a user's tendency to rate on a higher or lower scale than the average rater, while the item bias captures the extent of the item popularity. The user's trait vector p_u represents the user's preferences or “taste”. Similarly, the item's traits vector represent the item's latent characteristics. The dot-product $p_u^\top q_i$ is the personalization component which captures user's u affinity to item i.

A significant strength of MF models is their natural ability to easily incorporate additional information. For example, temporal dynamics and taxonomy components can be easily incorporated into the MF model [6]. In such models the prediction equation takes a more complex form. Without loss of generality, we will only discuss the basic model (equation 2). However, the more complex models have more parameters which are usually additive. Hence, the proposed reduction can be easily extended and applied to these model as well. In fact, we use one such complex MF model (described in [6]) for evaluating our proposed methods.

Training.

There are various techniques for training MF models. Generally a cost function is defined on the prediction error (e.g., RMSE) and optimization is followed by Stochastic Gradient Descent or an Alternating Least Squares algorithm. The reader is referred to [13] for more details on those learning techniques. The training produces estimates of the MF model parameters (the trait vectors and the biases). Given these parameters, we will demonstrate that the user's preference for any item can be reduced to a simple dot-product.

2.1 Reduction of RoR

RoR in a trained MF model involves finding the set of K items for a user u with maximum predicted ratings. Equation 2 implies that for a given user u, ordering the items is independent of μ and the user's bias b_u. Thus, we can ignore these parameters without affecting the preferred ordering of the items and obtain an effective rating:

$$\tilde{r}_{ui} = b_i + p_u^\top q_i. \tag{3}$$

It is important to note that this is only applicable during the RoR phase (after the training). Ignoring these components during the training will inevitably result in poor accuracy.

By appending the item bias to the item vector, the effective rating (equation 3) reduces to a simple dot-product:

$$\tilde{r}_{ui} = \tilde{p}_u^\top \tilde{q}_i = \|\tilde{p}_u\| \, \|\tilde{q}_i\| \cos\left(\theta_{\tilde{p}_u,\tilde{q}_i}\right), \tag{4}$$

where $\tilde{p}_u = [p_u^\top \;\; 1]^\top$, $\tilde{q}_i = [q_i^\top \;\; b_i]^\top$ and $\theta_{\tilde{p}_u,\tilde{q}_i}$ is the angle between $\tilde{p}_u$ and $\tilde{q}_i$ at the origin.

Equation 4 implies that for a given user $\tilde{p}_u$, the items ordering is independent of the norm $\|\tilde{p}_u\|$ and only depends on the user vector through the angle $\theta_{\tilde{p}_u,\tilde{q}_i}$. Hence, without loss of generality, we normalize the user vector to a unit vector $\bar{p}_u = \tilde{p}_u / \|\tilde{p}_u\|$ to further simplify equation 4 to:

$$\bar{r}_{ui} = \bar{p}_u^\top \tilde{q}_i = \|\tilde{q}_i\| \cos\left(\theta_{\bar{p}_u,\tilde{q}_i}\right). \tag{5}$$

Note that while we normalize the concatenated user vectors, the concatenated item vectors can not be normalized without

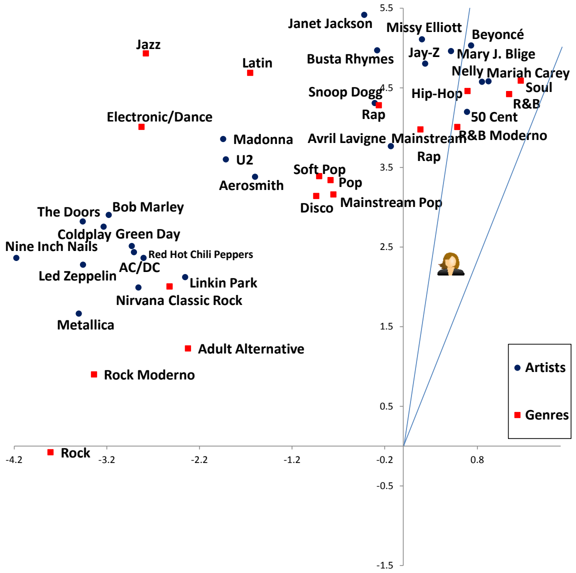


Figure 1: The most popular musical tracks and genres in the Yahoo! Music dataset are embedded into a 2-dimensional. The open cone suggests the region of highly preferred items for the user (her logo is at the center of the cone). Note how the learned embedding separates Rock and similar items from Hip-Hop and similar items. The low dimensionality (which is required for a visualization), causes a small number of items to be wrongly folded near less related items (e.g., Bob Marley).

loss of accuracy. If the item vectors were normalized, the RoR problem would have been reduced to the well studied nearest neighbor search (as explained in section 3). However, such a normalization will introduce a distortion on the balance between the original item trait vector and the item bias which constitute the concatenated vector $\tilde{q}_i$. This would evidently result in an incorrect solution.

Denoting the concatenated user and item vectors as p_u and q_i respectively, and the effective rating for the task of retrieval as r_{ui}, RoR reduces to the following task: Given a user query p_u, we want to find an item $q_i \in S$ such that:

$$p_u^\top q_i = \max_{q \in S} p_u^\top q \tag{6}$$

Hence the RoR task is equivalent to the problem of finding the best-match for a query in a set of points with respect to the dot-product (described in equation 1). A very simplistic visualization of this task is depicted in Figure 1. For the given user, the best recommendations (in this case songs) lie within the open cone around the user vector (maximizing the $\cos(\theta_{p_u,q_i})$ term) and are as far as possible from the origin (maximizing the $\|q_i\|$ term).

3. ALGORITHMS FOR FINDING BEST-MATCHES

Efficiently finding the best match using the dot-product (equation 6) appears to be very similar to much existing work in the literature. Finding the best match with respect to the Euclidean (or more generally L_p) distance is the widely studied problem of nearest-neighbor search in metric spaces [4]. The nearest-neighbor search problem (in metric space) can be solved approximately with the popular *Locality-sensitive hashing* (LSH) method [9]. LSH has been extended to other forms of similarity functions (as opposed to the distance as a dissimilarity function) like the cosine similarity [3]. In this section, we show that the problem stated in equation 6 is different from these existing problems.

Nearest-neighbor Search in Metric Space.

The problem of finding the nearest-neighbor in this setting is to find a point $q_i \in S$ for a query p_u such that:

$$\begin{aligned} q_i &= \arg\min_{q \in S} \|p_u - q\|^2 = \arg\max_{q \in S} \left(p_u^\top q - \|q\|^2/2\right) \\ &\neq \arg\max_{q \in S} p_u^\top q \text{ (unless } \|q\|^2 = const \quad \forall\, q \in S). \end{aligned}$$

If all the points in S are normalized to the same length, then the problem of finding the best match with respect to the dot-product is equivalent to the problem of nearest-neighbor search in any metric space. However, without this restriction, the two problems can yield very different answers.

Cosine similarity.

Finding the best match with respect to the cosine similarity is to find a point $q_i \in S$ for a query p_u such that

$$\begin{aligned} q_i &= \arg\max_{q \in S} p_u^\top q / (\|p_u\| \, \|q\|) = \arg\max_{q \in S} p_u^\top q / \|q\| \\ &\neq \arg\max_{q \in S} p_u^\top q \text{ (unless } \|q\| = const \quad \forall\, q \in S). \end{aligned}$$

As in the previous case, the best match with cosine similarity is the best match with dot-products if all the points in the set S are normalized to the same length. Under general conditions, the best matches with these two similarity functions can be very different.

Locality-sensitive Hashing.

LSH involves constructing hashing functions h which satisfy the following for any pair of points $q, p \in \mathbb{R}^D$:

$$\Pr[h(q) = h(p)] = \text{sim}(q, p), \tag{7}$$

where $\text{sim}(q,p) \in [0,1]$ is the similarity function of interest. For our situation, we can scale our dataset such that $\forall\ q \in S, \|q\| \leq 1$ and assume that the data is in the first quadrant (such as in non-negative matrix factorization models [19]). In that case, $\text{sim}(q,p) = q^\top p \in [0,1]$ is our similarity function of interest.

For any similarity function to admit a locality sensitive hash function family (as defined in equation 7), the distance function $\mathbf{d}(q,p) = 1 - \text{sim}(q,p)$ must satisfy the triangle inequality (Lemma 1 in [3]). However, the distance function $\mathbf{d}(q,p) = 1 - q^\top p$ does not satisfy the triangle inequality. Hence LSH cannot be applied to the dot-product similarity function even in restricted domains (the first quadrant).

3.1 Why is finding the maximum dot-products harder?

Unlike the distance functions in metric space, dot-products do not induce any form of triangle inequality (even under some assumptions as mentioned in the previous section). Moreover, this lack of any induced triangle inequality causes the similarity function induced by the dot-products to have no admissible family of locality sensitive hashing functions. Any modification to the similarity function to conform to widely used similarity functions (like Euclidean distance or Cosine-similarity) will create inaccurate results.

Moreover, dot-products lack the basic property of *coincidence* – the self similarity is highest. For example, the Euclidean distance of a point to itself is 0; the cosine-similarity of a point to itself is 1. The dot-product of a point q to itself is $\|q\|^2$. There can possibly be many other points v_i $(i = 1, 2, \ldots)$ in the set such that $q^\top v_i > \|q\|^2$.

Without any assumptions, this problem of obtaining the best match with respect to the dot-product is inherently harder than the previously addressed similar problems. This is possibly the reason why there is no existing work for this problem without any restrictions on the domain.

4. FAST EXACT RETRIEVAL USING METRIC TREES

In this section, we describe metric trees and develop a novel bound to use with a simple branch-and-bound algorithm to provide the first method to efficiently obtain the exact best-matches with respect to the dot-products.

4.1 Metric Trees

Metric trees [16] are binary space-partitioning trees that are widely used for the task of indexing datasets in Euclidean spaces. The space is partitioned into overlapping hyper-spheres (balls) containing the points (figure 2). We use a simple metric tree construction heuristic that tries to approximately pick a pair of pivot points farthest apart from each other [15][2], and splits the data by assigning points to their closest pivot. The tree T is built hierarchically and each node in the tree is defined by the mean of the data in that node (T.center) and the radius of the ball around the mean enclosing the points in the node (T.radius). The tree has leaves of size at most N_0. The splitting and the recursive tree construction algorithm is presented in Algorithms 1 & 2.

The tree is space efficient since every node only stores the indices of the item vectors instead of the item vectors themselves. Hence the matrix for the items is never duplicated. Another implementation optimization is that the vectors in the items' matrix are sorted in place (during the tree construction) such that all the items in the same node are arranged serially in the matrix. This avoids random memory access while accessing all the items in the same leaf node.

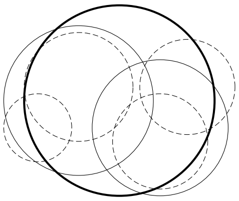


Figure 2: Metric-trees – note that while all the points in a child node lie also inside the parent ball, the child ball itself does not necessarily lie within the parent ball.

4.2 Branch-and-bound algorithm

Metric trees are used for efficient nearest neighbor search and are fairly scalable in moderately high dimensions [15, 14]. The search employs a depth-first branch-and-bound algorithm. A nearest-neighbor query is answered by traversing the tree in a depth-first manner– going down the node closer to the query first and bounding the minimum possible distance to items in other branches with the triangle-inequality. If this branch is farther away than the current neighbor candidate, the branch is removed from computation.

Algorithm 1 MakeMetricTreeSplit(Data S)

```
Pick a random point $\mathbf{x} \in S$
$A \leftarrow \arg\max_{\mathbf{x}' \in S} \|\mathbf{x} - \mathbf{x}'\|$
$B \leftarrow \arg\max_{\mathbf{x}' \in S} \|A - \mathbf{x}'\|$
$\mathbf{w} \leftarrow (B - A)$
$b \leftarrow -\frac{1}{2}\left(\|B\|^2 - \|A\|^2\right)$
return $(\mathbf{w}, b)$
```

Algorithm 2 MakeMetricTree(Set of items S)

```
Input – Set $S$
Output – Tree $Q$
$Q.S \leftarrow S$
$Q$.center $\leftarrow$ mean($S$)
$Q$.radius $\leftarrow \max_{q_i \in S} \|T.\text{center} - q_i\|$
if $|S| \leq N_0$ then
    // Leaf node
    return $Q$
else
    // else split the set
    $(\mathbf{w}, b) \leftarrow$ MakeMetricTreeSplit($S$)
    $S_l \leftarrow \{q_i \in S \colon \mathbf{w}^\top q_i + b \leq 0\}$
    $S_r \leftarrow S \setminus S_l$
    $Q$.left $\leftarrow$ MakeMetricTree($S_l$)
    $Q$.right $\leftarrow$ MakeMetricTree($S_r$)
    return $Q$
end if
```

Figure 3: Metric-tree Construction: The object $Q.S$ denotes the set of items in the node Q, Q.center denotes the Euclidean mean of the items in the node Q and Q.radius denotes the minimum radius of the ball centered around Q.center enclosing all the items in the node Q.

Since the triangle inequality does not hold for the dot-product, we present a novel analytical upper bound for the maximum possible dot-product of a user vectors with points (in this case, items) in a ball. We then employ a similar branch-and-bound algorithm for the purposes of searching for the K-highest dot-products (as opposed to the minimum pairwise distance in K-nearest-neighbor search).

4.2.1 *Bounding with a ball*

Let $B^r_{q_c}$ be the ball of items centered around q_c with radius r. Suppose that q^* is the best possible recommendation in the ball $B^r_{q_c}$ for the user represented by the vector p_u, and r^* be the Euclidean distance between the ball center q_c and the best possible recommendation q^* (by definition, $r^* \leq r$). Let θ be the angle between the vector $\vec{q_c}$ and the vector $\overrightarrow{q_c q^*}$, θ_{p_u,q_c} and θ_{q^*,q_c} be the angles between the vector $\vec{q_c}$ and vectors $\vec{p_u}$ and $\vec{q^*}$ respectively (see figure 4). The distance of q^* from q_c is $r^* \sin\theta$ and the length of the projection of q^* onto q_c is $\|q_c\| + r^* \cos\theta$. Therefore we have:

$$\|q^*\| = \sqrt{(\|q_c\| + r^* \cos\theta)^2 + (r^* \sin\theta)^2}, \qquad (8)$$

$$\cos\theta_{q^*,q_c} = \frac{\|q_c\| + r^* \cos\theta}{\|q^*\|}, \sin\theta_{q^*,q_c} = \frac{r^* \sin\theta}{\|q^*\|}. \qquad (9)$$

[2] The intuition behind this heuristic is that these 2 points farthest from each other might lie in the principal direction (the direction of the principal eigenvector of the data).

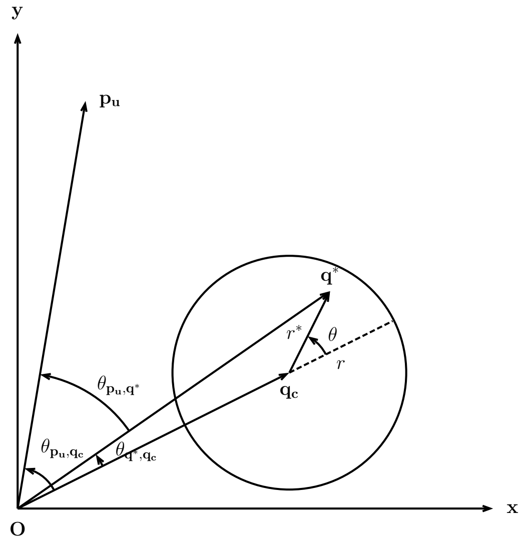

y
$\mathbf{p_u}$
$\mathbf{q^*}$
r^*
θ
r
$\mathbf{q_c}$
$\theta_{\mathbf{p_u},\mathbf{q^*}}$
$\theta_{\mathbf{p_u},\mathbf{q_c}}$
$\theta_{\mathbf{q^*},\mathbf{q_c}}$
x
O

Figure 4: Bounding with a ball

Let θ_{p_u,q^*} be the angle between the vectors $\vec{p_u}$ and $\vec{q^*}$. This gives the following inequality regarding the angle between the user and the best possible recommendation (we assume that the angles lie in the range of $[-\pi, +\pi]$ instead of the usual $[0, 2\pi]$) :

$$|\theta_{p_u,q^*}| \geq |\theta_{p_u,q_c} - \theta_{q^*,q_c}|,$$

which implies

$$\cos\theta_{p_u,q^*} \leq \cos(\theta_{p_u,q_c} - \theta_{q^*,q_c}), \tag{10}$$

since $\cos(\cdot)$ is monotonically decreasing in the range $[0, \pi]$. Using this equality we obtain the following bound for the highest possible affinity between the user and any item within that ball:

$$\begin{aligned} \max_{q_i \in B^r_{q_c}} p_u^\top q_i &= p_u^\top q^* (\text{by assumption}) \\ &= \|p_u\| \; \|q^*\| \cos\theta_{p_u,q^*} \\ &\leq \|p_u\| \; \|q^*\| \cos(\theta_{p_u,q_c} - \theta_{q^*,q_c}), \end{aligned}$$

where the last inequality follows from equation 10. Substituting equations 8 & 9 in the above inequality, we have

$$\begin{aligned} \max_{q_i \in B^r_{q_c}} p_u^\top q_i &\leq \|p_u\| \left(\cos\theta_{p_u,q_c} (\|q_c\| + r^* \cos\theta)\right. \\ &\quad \left. + \sin\theta_{p_u,q_c} (r^* \sin\theta)\right) \\ &\leq ||p_u|| \max_\theta \left(\cos\theta_{p_u,q_c} (\|q_c\| + r^* \cos\theta)\right. \\ &\quad \left. + \sin\theta_{p_u,q_c} (r^* \sin\theta)\right) \\ &= ||p_u|| \left(\cos\theta_{p_u,q_c} (\|q_c\| + r^* \cos\theta_{p_u,q_c})\right. \\ &\quad \left. + \sin\theta_{p_u,q_c} (r^* \sin\theta_{p_u,q_c})\right) \\ &\leq ||p_u|| \left(\cos\theta_{p_u,q_c} (\|q_c\| + r \cos\theta_{p_u,q_c})\right. \\ &\quad \left. + \sin\theta_{p_u,q_c} (r \sin\theta_{p_u,q_c})\right) \\ &\quad (\text{since } r^* \leq r). \end{aligned}$$

The second inequality comes from the definition of maximum, and the next equality comes from maximizing over θ giving us the optimal value for $\theta = \theta_{p_u,q_c}$. Simplifying the final inequality gives us the following upper bound:

$$\max_{q_i \in B^r_{q_c}} p_u^\top q_i \leq p_u^\top q_c + r\,\|p_u\|\,. \tag{11}$$

Algorithm 3 SearchMetricTree(User p_u, Item Tree Node Q)

```
if $p_u$.ub < $p_u^\top Q$.center + $Q$.radius · $\|p_u\|$ then
  // This node has potential
  if isLeaf(Q) then
    for each $q_i \in Q.S$ do
      if $p_u^\top q_i > p_u$.ub then
        $q' \leftarrow \arg\min_{q \in p_u.\text{candidates}} p_u^\top q$
        $p_u$.candidates ← $\{p_u.\text{candidates} \setminus \{q'\}\} \cup \{q_i\}$
        $p_u$.ub ← $\min_{q \in p_u.\text{candidates}} p_u^\top q$
      end if
    end for
  else
    // best depth first traversal
    $I_l \leftarrow p_u^\top Q$.left.center; $I_r \leftarrow p_u^\top Q$.right.center;
    if $I_l \leq I_r$ then
      SearchMetricTree($p_u$, $Q$.right);
      SearchMetricTree($p_u$, $Q$.left);
    else
      SearchMetricTree($p_u$, $Q$.left);
      SearchMetricTree($p_u$, $Q$.right);
    end if
  end if
end if
// Else the node is pruned from computation
return;
```

Algorithm 4 FindExactRecommendations(User p_u, Item Tree Node Q)

```
$p_u$.ub ← 0;
$p_u$.candidates ← $\emptyset$;
SearchMetricTree($p_u$, $Q$);
return $p_u$.candidates;
```

Figure 5: Metric-tree Search: The object p_u.candidates contains the set of current best K candidate items and p_u.ub denotes the lowest affinity between the user and its current best candidates.

4.2.2 The Algorithm

Using this upper bound (11) for the maximum possible dot-product, we present the depth-first branch-and-bound algorithm to search for the K-highest dot-products in Algorithm 3. The algorithm begins at the root of the tree of items. At each subsequent step, the algorithm is at a tree node. Using the bound in equation 11, the algorithm checks if the best possible item in this node is any better than the current best candidates for the user. 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 candidates within the leaf with the simple naive search. This algorithm ensures that the exact solution (i.e., the best candidates) is returned by the end of the algorithm.

Theoretical Runtime Bounds.

We do not have any runtime guarantees for the algorithm presented in this section. However, we can conjecture possible runtime bounds. If the metric-tree constructed with Algorithm 2 has a depth of $O(\log|Q|)$ (where Q is the set of items), then the runtime bound for the construction of the tree is $O(D|Q|\log|Q|)$ (since you only require $O(D|Q|)$

operations at each level of the tree and D is the dimensionality of the data). During the tree-search algorithm (Alg. 4), let us assume that the user visits L leaves. If L is much smaller and in fact independent of the number of items $|Q|$, we can say that the runtime bound for the search process for a single user is $O(DL \log |Q|)$. However, if L depends on $|Q|$ as well, then the best possible runtime bound is $O(D|Q|)$.

Since algorithm 2 does not enforce that the splits be balanced, it is quite possible that the depth of the tree might end up being $O(|Q|)$, in which case, the worst case runtime for the search process is $O(D|Q|)$. However, in practice, the tree depths have been seen to be way less than $O(|Q|)$.

5. FAST APPROXIMATE RETRIEVAL BY CLUSTERING USERS

The efficiency of the exact algorithm can be limited, and some applications may require even faster retrieval while allowing for some suboptimal recommendations. To this end, we propose a scheme to cluster the users into *cones* of similar "taste", pre-compute the recommendations for the cone centers (representative user tastes), and use these recommendations as approximate recommendations for incoming users with tastes similar to some existing user cluster.

Equation 5 specifies that the user preferences depend only on the angle (direction) of the corresponding user vectors. The smoothness of the cosine function implies that two users with vectors in similar directions will have very similar preferences. Hence, we partition the space into cones that aggregate users with similar taste. Let P_c be a set of cone centers where each $p_c \in P_c$ is a unit vector. The direction of p_c is the taste of the cone which can be used to pre-compute recommendations for the users in that cone.

For a new user query p_u, its best cone p_c^* is:

$$p_c^* \leftarrow \arg\max_{p_c \in P_c} p_u^\top p_c$$

After finding p_c^* we retrieve the pre-computed recommendations of p_c^* as the approximated recommendations for p_u. Figure 6 depicts a user's vector p_u and its best cone's vector p_c^*. If Δ, the angle between p_u and p_c^*, is small enough, then the approximated recommendations will be close to the optimal recommendations. The speedup is achieved since the number of cones is much smaller than the number of users or items, thus finding p_c^* is significantly easier than computing the exact recommendations for each user .

The approximation is controlled by using an upper bound on the relative approximation error in terms of Δ (this is presented in section 5.1). This bound evaluates the quality of the approximation. By defining a threshold T_r on the maximum acceptable error, we adaptively accept the pre-calculated approximate results when the error is below the threshold, or compute the exact results otherwise. The details of the approximate RoR algorithm are given in figure 7. The threshold T_r introduces a tradeoff between speedup and accuracy where maximal speedup is achieved when $T_r = \infty$.

Choosing cones.

In general, one would like to choose a set of user clusters (cones) which appropriately fits the distribution of possible queries. This can be efficiently achieved by spherical clustering of the user vectors. Spherical clustering defines groups of users with similar preferences or taste. Unit vectors in the direction of the clusters' centers define the cone centers.

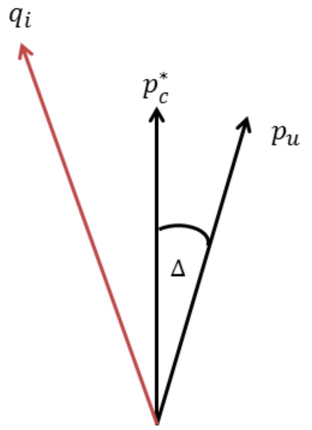


Figure 6: Here p_u is the user's vector, p_c^* is the cone's central direction, q_i is the item's vector and Δ is the angle between p_u^* and p_c.

Algorithm 5 PrepareCones(Users P, Items S)

```
Input – Set $P$, $S$
Output – User Cones $P_c$, Tree $Q$
$P_c$ = ChooseCones($P$);
$Q$ = MakeMetricTree($S$);
for all $p_c \in P_c$ do
    $p_c$.candidates = FindExactRecommendations($p_c, Q$);
end for
```

Algorithm 6 FindApproxRecommendations(User p_u, User Cones P_c, Threshold T_r, Item Tree Q)

```
$p_c^* \leftarrow \arg\max_{p_c \in P_c} p_u^\top p_c$;
$ErrorBound$ = ComputeErrorBound($p_c^*, p_u$);
if $ErrorBound \leq T_r$ then
    return $p_c^*$.candidates;
else
    return FindExactRecommendations($p_u, Q$);
end if
```

Figure 7: Approximate RoR: The subroutine *ChooseCones* chooses a set of cones that fits the set of user vectors P in the dataset. Then, the optimal recommendations are computed for each cone's center using the metric tree of section 4.
In *FindApproxRecommendations*, the subroutine *ComputeErrorBound* computes the error bound according to equation 14. The approximated recommendations are used for every query with an error bound below the error threshold T_r, otherwise exact recommendations are computed.

We chose spherical clustering because of its computational efficiency. Furthermore, the clustering already assigns the user vectors into cones and there is no need to search for the best matching cone for the existing users.

A requirement for a good clustering is that $\Delta < \frac{\pi}{2}$. Otherwise the dot-product between the user and an item in the direction of the cone's center can be negative, which implies that the user does not like the items that fit the cone's vector. Note that clustering assumes the presence of groups of users common tastes. This is a very natural assumption in every collaborative filtering algorithm.

5.1 Relative Error Bound

In this subsection we present a theoretical error bound on the relative approximation error for any user. This is used by the adaptive algorithm to control the approximation error.

The vector q_i in figure 6 depicts an item's vector that was chosen as an optimal recommendation based on the cluster's center p_c^*. Intuitively, as Δ decreases, the approximation error should decrease as well. We define the approximation error $err = exp - real$ as the difference between the expected rating based on the cluster $exp = q_i^\top p_c^*$ and the real dot-product with the user's trait vector $real = q_i^\top p_u$. The relative error is then:

$$\frac{err}{exp} = \frac{exp - real}{exp} = 1 - \frac{real}{exp} \tag{12}$$

We assume here that for every cone $exp > 0$; otherwise it means that there are no fitting recommendation for that cone, which is very unlikely and we never encountered this[3].

Since we want the worst-case bound, we ignore the case where $real > exp$ since this situation means that the affinity between p_u and q_i is better than expected. Hence, assuming that $real < exp$, we have the following:

$$\frac{real}{exp} = \frac{\|q_i\| \cos\left(\theta_{p_u,q_i}\right)}{\|q_i\| \cos\left(\theta_{p_c^*,q_i}\right)} \geq \frac{\cos\left(\theta_{p_c^*,q_i} + \Delta\right)}{\cos\left(\theta_{p_c^*,q_i}\right)} \tag{13}$$

The inequality follows from the fact that $\theta_{p_c^*,q_i} \leq \frac{\pi}{2}$ (because $exp \geq 0$) and $\Delta \leq \frac{\pi}{2}$ (a requirement of the clustering). Since $\cos(\cdot)$ is monotonically decreasing in the range $[0, \pi]$, we get $\cos\left(\theta_{p_u,q_i}\right) \geq \cos\left(\theta_{p_c^*,q_i} + \Delta\right)$. Substituting (13) into (12) we get the following upper bound on the relative error:

$$\frac{err}{exp} \leq 1 - \frac{\cos\left(\theta_{p_c^*,q_i} + \Delta\right)}{\cos\left(\theta_{p_c^*,q_i}\right)} \tag{14}$$

Note that this bound is a tight bound. Namely, when $\Delta \to 0$ we get $err \to 0$.

6. EXPERIMENTS

We begin this section by presenting the datasets and the evaluation metric used. Then we present the results of the exact RoR algorithm (section 4). In the following subsection, we present the performance of the approximate RoR method (section 5), demonstrating its efficiency-error trade-off. Finally, as a thought experiment, we also present the inaccuracies introduced by using existing best-match algorithms (nearest-neighbor search in Euclidean space and best-match with respect to cosine similarity) for the task of RoR.

Datasets.

We used the following publicly available datasets:

1. MovieLens – It consists of 1,000,206 ratings of 3,952 movies by 6,040 users. Ratings are integers in the range 1-5, and the dataset is 95.81% sparse.

2. Netflix [2] – It consists of 100,480,507 ratings of 17,770 movies by 480,189 users. The ratings in Netflix are on a scale of 1-5 as well, and the dataset is 98.82% sparse.

3. Yahoo! Music [8] – This dataset is the largest of the three consisting of 252,800,275 ratings of 624,961 music items by 1,000,990 users. The ratings are on a scale of 0-100 and the dataset is 99.96% sparse.

Currently the Yahoo! Music dataset is the largest publicly available collaborative filtering dataset. Both our algorithms perform best on this dataset. In fact, in most of our evaluations the results seems to improve with the size of the dataset. This is expected as overhead times become negligible when the number of queries (users) increase. All the above datasets were in fact sampled from real datasets which were possibly much larger. It is therefore likely that the results presented in this paper will further improve when the proposed algorithms are implemented in real world systems.

For the MovieLens and Netflix datasets, we built and trained a basic MF model (equation 2) using stochastic gradient descent minimization of the mean squared error. For the Yahoo! Music dataset, we used the model presented in [6] that incorporates music taxonomy and temporal effects. All models have 50-dimensional vectors to represent the user and item traits. The root mean squared errors of these three models were 0.839 in MovieLens, 0.899 in Netflix, and 22.592 in Yahoo! Music.

We quantify the improvement of an algorithm A over another (baseline) algorithm A_0 by the following term:

$$\text{Speedup}_{A_0}(A) = \frac{\text{Time taken by Algorithm } A_0}{\text{Time taken by Algorithm } A}. \tag{15}$$

Since there are no efficient search algorithms for maximum dot-products, our baseline is a naive algorithm that searches over all items to find the best recommendations for every user. We denote by T_{naive} the time taken by the naive algorithm. It is obvious that

$$T_{naive} = \Theta(\#USERS \times \#ITEMS \times D),$$

where D is the dimensionality of the vector (here $D = 50$).

As expected, the naive algorithm is extremely time consuming. For example, the baseline execution time for retrieving optimal recommendations for the Yahoo! Music dataset is 135.1 hours[4] (over 5 days). The mean latency for a single user query is 0.482 seconds. Using our proposed combined method (figure 7), we achieve up to ×258.08 speedup, which is equivalent to just 31.4 minutes for the entire computation or an average single user latency of 1.87 milliseconds. It is important to note that while the overall computation time can also be reduced by means of parallelization, the latency for a single user might be harder to improve upon.

Implementation Details.

We used the *Cluto* clustering toolkit [10] for spherical clustering of the user vectors. We used 500, 1000, and 2000 clusters (cones) for the MovieLens, Netflix and Yahoo! Music respectively, because these values showed a good balance between performance and speedup. In Alg. 2, we used $N_0 = 2$ in all our experiments. In general, these parameters can be optimized using a cross-validation process.

6.1 Exact RoR

The time taken by the exact algorithm of Section 4 can be broken up into two parts as follows:

$$T_{exact} = T_{tree\ building} + T_{tree\ search},$$

where $T_{tree\ building}$ is the time taken by Alg. 2 to build the tree on the set of item vectors, and $T_{tree\ search}$ is the total time taken by all the users to find their respective best

[3] Even if $exp < 0$ it is still possible to bound the error by following a very similar process to the one shown here.

[4] Using an Intel Xeon (E7320) CPU running at 2.13GHz

Dataset	$K=1$	$K=5$	$K=10$	$K=50$
MovieLens	3.01	1.82	1.73	1.21
Netflix	2.87	2.39	1.95	1.31
Yahoo! Music	7.26	5.25	4.7	3.01

Table 1: Speedups of Alg. 3 over naive search for different number of top recommendations (K).

recommendations using Alg. 3. The speedup is therefore:

$$\text{Speedup}_{naive}(exact) = \frac{T_{naive}}{T_{exact}}. \tag{16}$$

We present the speedups obtained for different numbers of top recommendations in Table 1. The results indicate that the exact algorithm can be up to ×7 faster than the naive algorithm. Another advantage of this method is its space efficiency – only the tree (which consists solely of pointers) has to be stored. The complete ($\#USERS \times \#ITEMS$) user-preference matrix does not have to be stored and the recommendations for a user can be obtained when required.

An important thing to note is that the tree-building task is extremely time efficient – for example, for the Yahoo! Music dataset, the time taken to build the metric-tree on the set of items (of size 624961×50) was less than 16 seconds (the time required to load the whole data into memory took more than 40 seconds!). The tree-building process is a one-time cost which is amortized by the more expensive tree-search process. Moreover, new items can be easily added to this metric-tree index[5]. Nevertheless, we include the tree-building times in our computation for completeness.

It is important to note that the search time increases with K (the number of top recommendations returned). This is because the bound for the best recommendations for the user (p_u.ub in Alg. 3) becomes smaller with increasing K (Line 8 in Alg. 3). This increases the number of nodes that have potential (Line 1 in Alg. 3), hence also increasing the number of leaves finally visited.

However, some applications may require more than just the top 50 items. In that case, the tree-based exact search does not provide any significant improvement over the naive algorithm. Therefore, we present further improvements in computational performance in the next subsection with the proposed approximate algorithm.

6.2 Approximate RoR

The time taken by the approximate algorithm of Section 5 can be broken up into four parts as follows:

$$\begin{aligned} T_{approx} = {} & T_{clustering} + T_{tree\ building} \\ & + T_{search\ cones} + T_{search\ queries}, \end{aligned}$$

where $T_{clustering}$ is the time taken by the clustering algorithm, $T_{tree\ building}$ is the metric-tree-construction time, $T_{search\ cones}$ is the search time for optimal recommendations for all the cones and $T_{search\ queries}$ is the time taken to compute exact recommendations for queries that are above the threshold. The speedup of the approximate algorithm is:

$$\text{Speedup}_{naive}(approx) = \frac{T_{naive}}{T_{approx}}. \tag{17}$$

We define two terms to quantify the quality of the top K recommendations retrieved by the approximated method. The first quantity ($Precision$) denotes how similar the approximate recommendations are to the actual top K recommendations (which are retrieved by the naive approach):

MovieLens

Threshold	0.25	0.5	0.75	∞
K=1	x2.49	x7.26	x9.02	x9.25
K=5	x1.6	x5.68	x7.66	x7.94
K=10	x1.52	x5.5	x7.49	x7.78
K=50	x0.89	x3.82	x7.73	x6.04

Netflix

Threshold	0.25	0.5	0.75	∞
K=1	x2.69	x5.65	x12.61	x17.29
K=5	x2.48	x5.3	x12.27	x17.26
K=10	x1.93	x4.29	x11.15	x17.18
K=50	x1.19	x2.81	x8.89	x16.97
K=500	x1.04	x2.5	x8.28	x16.89

Yahoo! Music

Threshold	0.25	0.5	0.75	∞
K=1	x10.49	x19.7	x150.87	x258.08
K=5	x7.67	x14.54	x128.46	x251.27
K=10	x6.88	x13.08	x120.77	x248.46
K=50	x4.45	x8.53	x91.78	x234.6
K=500	x1.49	x2.89	x39.1	x178.64

Table 2: Speedups of Alg. 6 over the naive algorithm for different values of K and the error threshold.

$$Precision(K) \triangleq \text{mean}_u \left\{ \frac{|L_{rec}(u) \cap L_{opt}(u)|}{K} \right\}, \tag{18}$$

where $L_{rec}(u)$ and $L_{opt}(u)$ are the lists of the top K approximate and the top K optimal recommendations for the user u, respectively. Our evaluation metrics only care about the items at the top of the approximated and optimal lists ($L_{rec}(u)$ and $L_{opt}(u)$). In that case there is no real meaning to compute $Recall$ because its natural definition would be identical to the $Precision$.

In addition, we define a secondary metric ($MedianRank$) which denotes the preference of the approximated recommendations with respect to the rest of the items:

$$MedianRank(K) \triangleq \text{median}\left\{\cup_u Rank(L_{rec}(u))\right\}, \tag{19}$$

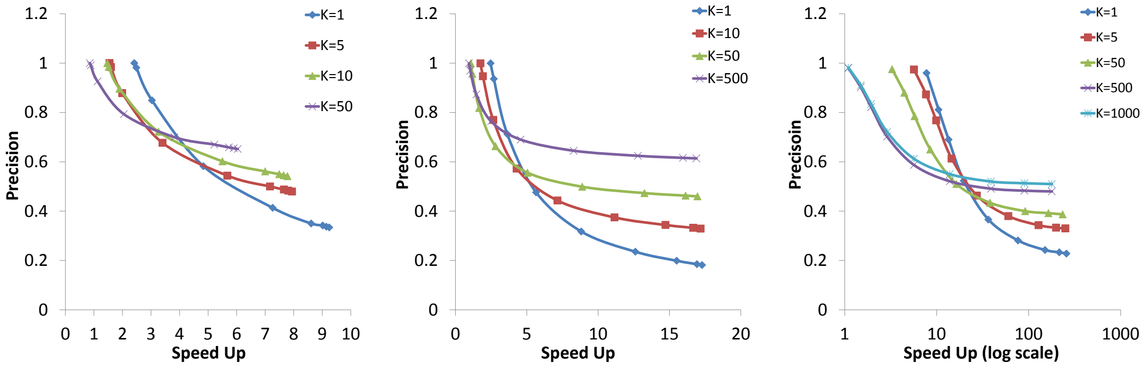
where the function $Rank(L(u))$ returns a list of the optimal ranks for the items in $L(u)$ for user u (for example, $Rank(L_{opt}(u)) = \{1, 2, \ldots, K-1, K\}$).

A high value for $Precision$ implies that the approximate recommendations are very similar to the optimal recommendations, and a low value of $MedianRank$ implies that the approximate recommendations are highly preferred by the users. In many practical applications, it is very likely to have a low value for $Precision$ as well as for $MedianRank$. This implies that the items recommended by the approximate algorithm are generally different from the optimal items for the users, but the items recommended are still very highly preferred by the users.

The speedups of the approximate algorithm for different values of the error bound threshold are summarized in Table 2. The results indicate that the approximate RoR method can be up to ×258 faster than the naive approach. The approximation quality for different levels of speedup is depicted in figures 8 & 9.

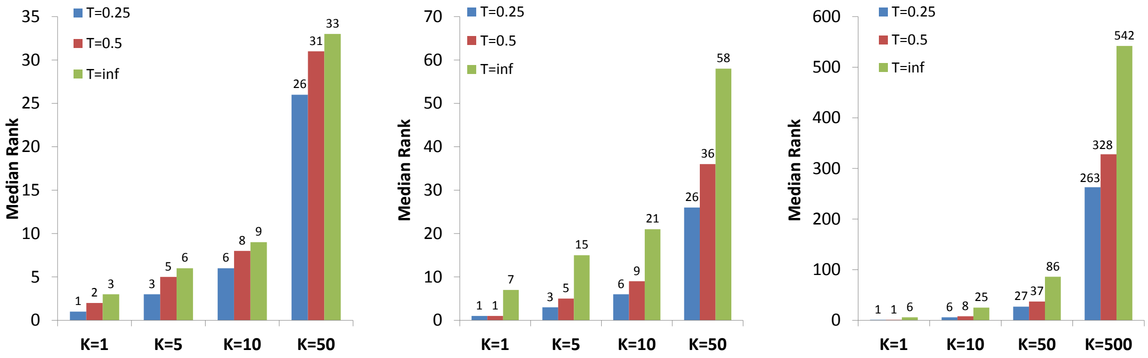
Figure 8 shows the tradeoff between precision and speedup achieved by using different values of the error bound threshold. When the threshold is high, the approximated result is less likely to be rejected. In this case, the precision is lower, speedup is higher, and performance is better for higher values of K. The latter is a result of the fact that precision

[5] Efficient item insertion is inherent to tree data structures.



(a) MovieLens (b) Netflix (c) Yahoo! Music

Figure 8: Precision(K) of L_{rec} vs. speedup of the approximate algorithm. The error bound threshold defines a tradeoff between high precision to high speedup. We used higher values of K for datasets with more items.



(a) MovieLens (b) Netflix (c) Yahoo! Music

Figure 9: $MedianRank(K)$ of the adaptive algorithm for different values of the error bound threshold. Speedup values can be retrieved from table 2.

in a finite set is easier to achieve as K is higher. When the threshold is low, the approximated result is more likely to be rejected. In this case, the precision is higher, speedup is lower, and performance is better when K is lower. The latter is a result of the fact that we are more likely to fall back to using the metric tree and the fact that the tree performs worse on higher values of K (as explained earlier).

Figure 9 presents the $MedianRank$ for different values of K and different values of the error bound threshold. Speedup values can be retrieved from table 2. We see that even when $Precison$ values are low (e.g., when $T_r = \infty$) the $MedianRank$ values are also relatively low, which indicate that the approximated recommendations are still highly preferred by the users.

6.3 Existing Best-Match Algorithms

Dataset	$K = 1$	$K = 5$	$K = 10$	$K = 50$	$K = 100$
MovieLens	0.4	0.54	0.59	0.72	0.77
Netflix	0.19	0.24	0.28	0.35	0.39
Y! Music	0.055	0.08	0.08	0.112	0.133

Table 3: Precision of the top (K) best matches with respect to the l_2 distance

In this subsection we find the top recommendations for a user with respect to the Euclidean (l_2) distance and with respect to the cosine similarity. The first returns the K items closest to the user (in terms of the l_2 distance), and the second returns the K items making the smallest angles with the user at the origin (hence returning best matches with respect to the cosine similarity). The reason for this experiment is to demonstrate that existing nearest-neighbor search algorithms (like LSH) cannot be applied directly to the task of RoR in the *existing MF framework* without introducing high levels of error.

Dataset	$K = 1$	$K = 5$	$K = 10$	$K = 50$	$K = 100$
MovieLens	0.05	0.12	0.16	0.35	0.46
Netflix	0.14	0.24	0.31	0.48	0.56
Y! Music	0.004	0.01	0.014	0.033	0.047

Table 4: Precision of the top (K) best matches with respect to the cosine similarity

Tables 3 & 4 report the precision of the exact best-matches obtained with respect to Euclidean distance and cosine similarity respectively. As expected from our discussion in section 3, the precision of these results are very low (especially on the larger Yahoo! Music dataset). Contrasting these numbers to the precision of the approximate solutions obtained from Alg. 6 (figure 8), we see that our approximate algorithm performs as accurately (if not better) with significant amount of speedup. For example, for the Yahoo! Music data set with $K = 50$, the best-matches with l_2 distance and cosine similarity have a precision of 0.112 and 0.033 respectively. In contrast, our proposed algorithm shows a speedup of about ×200 while achieving a precision level of around 0.4 (figure 8(c)).

It is important to note that these returned recommendations in both cases (l_2 distance and cosine-similarity) are the *exact best-matches with respect to their corresponding sense of similarity.* If the exact results are so inaccurate (in terms of recommendation quality), it is hard to expect good results once approximate techniques for these best-match problems like LSH is used. This indicates that any form of modification done to the RoR task in MF framework (equation 1 and hence equation 6) to fit into existing best-matching problems can introduce a high level of inaccuracies.

7. CONCLUSIONS

In this paper we address the problem of efficient retrieval of recommendations (RoR) within the MF framework. This problem is inherent in a myriad of online services and requires added attention with the current influx of users (and items) on the internet. The RoR task in MF frameworks can be formulated as finding best matches with respect to the dot-product similarity measure. However, there are no known solutions to this problem. We thus present an exact and an approximate novel algorithms to improve the scalability of this task. Efficient algorithms to find the maximum dot-product can possibly have impacts beyond the realm of collaborative filtering.

The exact method uses an existing indexing scheme to index the set of items, and the branch-and-bound algorithm with a novel bounding scheme to provide significant speedup (over ×7 faster) over the naive algorithm, while having minimal space requirements. It can be easily adapted to include new items (or new users) into the system. However, being an exact algorithm, it shows limited improvement in computational performance. Hence we relax the problem of RoR and present an approximate algorithm based on the novel idea of grouping the users using spherical-cones. The method subsequently stores the best recommendations of each of the cone centers as the approximate best recommendations for all the users within that cluster. This method shows much better scalability (up to ×258 speedup) with the trade-off of deviating from the otimal list of recommendations. However, we demonstrate that even when precision is low, the approximated items are still highly preferred by the users.

MF based models have demonstrated impressive performance in terms of scalability at training time as well as predictive accuracy. However, less accurate algorithms are often used in large scale web-services. This may be attributed to the computational bottleneck of retrieval of the recommendations. Long latency times are unacceptable in online services, and pre-computing recommendations to all the users is expensive in terms of computational time as well as storage. The methods presented in this paper alleviate this last obstacle, making the MF framework more approachable to large scale recommender-systems. This paper also gives the system's architects a choice of an exact algorithm with significant but limited scalability or an approximate algorithm with a favorable trade-off between quality and scalability.

The problem of fast RoR discussed in this paper inspired the solution to the general problem of fast maximum inner-product search [17]. Possible extensions of this work will be to develop approximate algorithms with user-specified bounded approximation. For example, the system can approximate the retrieval task to obtain any K-recommendations from among the best τ-recommendations where $K < \tau$ (similar to the approximation of the nearest-neighbor search problem in [18]).

8. REFERENCES

[1] R. M. Bell and Y. Koren. Lessons from the netflix prize challenge. *SIGKDD Explor. Newsl.*, 2007.

[2] J. Bennett and S. Lanning. The netflix prize. In *Proc. KDD Cup and Workshop*, 2007.

[3] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, 2002.

[4] K. Clarkson. Nearest-neighbor searching and metric space dimensions. *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, 2006.

[5] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, 2007.

[6] G. Dror, N. Koenigstein, and Y. Koren. Yahoo! music recommendations: Modeling music ratings with temporal dynamics and item taxonomy. In *Proc. 5th ACM Conference on Recommender Systems*, 2011.

[7] G. Dror, N. Koenigstein, and Y. Koren. Web scale media recommendation systems. *Proceedings of the IEEE*, pages 1–15, 2012.

[8] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The yahoo! music dataset and kdd-cup'11. *Journal Of Machine Learning Research*, 17:1–12, 2011.

[9] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*, 1998.

[10] G. Karypis. CLUTO a clustering toolkit. Technical report, Dept. of Computer Science, University of Minnesota, 2002.

[11] M. Khoshneshin and W. N. Street. Collaborative filtering via euclidean embedding. In *Proceedings of the fourth ACM conference on Recommender systems*, 2010.

[12] N. Koenigstein, N. Nice, U. Paquet, and N. Schleyen. The xbox recommender system. In *Proc. 6th ACM Conference on Recommender Systems*, 2012.

[13] Y. Koren, R. M. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 2009.

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

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

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

[17] P. Ram and A. Gray. Maximum inner-product search using cone trees. In *SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 2012.

[18] 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 Information Processing Systems 22.* 2009.

[19] S. Zhang, W. Wang, J. Ford, and F. Makedon. Learning from incomplete ratings using non-negative matrix factorization. In *Proceedings of the Sixth SIAM International Conference on Data Mining*, 2006.