TopRecs: Top-k Algorithms for Item-based Collaborative Filtering

Mohammad Khabbaz
University of British Columbia
Department of Computer Science
mkhabbaz@cs.ubc.ca

Laks V.S. Lakshmanan
University of British Columbia
Department of Computer Science
laks@cs.ubc.ca

ABSTRACT

Recommender systems help users find their items of interest from large data collections with little effort. Collaborative filtering (CF) is one of the most popular approaches for making recommendations. While significant work has been done on improving accuracy of CF methods, some of the most popular CF approaches are limited in terms of scalability and efficiency. The size of data in modern recommender systems is growing rapidly in terms of both new users and items and new ratings. Item-based recommendation is one of the CF approaches used widely in practice. It computes and uses an item-item similarity matrix in order to predict unknown ratings. Previous works on item-based CF method confirm its usefulness in providing high quality top-k results. In this paper, we design a scalable algorithm for top-k recommendations using this method, while meeting high accuracy requirements. We achieve this by probabilistic modeling of the similarity matrix. A unique challenge here is that the ratings that are aggregated to produce the aggregate predicted score for a user should be obtained from different lists for different candidate items and the aggregate function is non-monotone. We propose a layered architecture for CF systems that facilitates computation of the most relevant items for a given user. We design efficient top-k algorithms and data structures in order to achieve high scalability. Our algorithm is based on abstracting the key computation of a CF algorithm in terms of two operations – *probe* and *explore*. The algorithm uses a cost-based optimization whereby we express the overall cost as a function of a similarity threshold and determine its optimal value for minimizing the cost. We empirically evaluate our theoretical results on large real world datasets. Our experiments show our approach achieves scalability and accuracy at the same time.

1. INTRODUCTION

The number of items offered by modern information systems is growing rapidly. Browsing large collections and finding the right item of interest is not a trivial task for typical users. Recommender systems help users find their items of interest with little effort, through personalized recommendations. They take advantage of ratings by users on items they have previously experienced and make predictions. Collaborative filtering (CF) has gained significant attention as the most popular recommendation paradigm [1]. CF approaches take as input a huge sparse matrix, consisting of ratings of users on items. They output the most relevant (top-k) items to the current or active user by predicting her unknown ratings on candidate items. Most of the previous research on collaborative filtering has been concerned with improving the accuracy of predictions. Clearly, *besides accuracy, scalability is an important concern*, given the ever growing number of users and items in today's recommender systems. An example of such a system is Google News which has millions of visitors every day and a growing number of news feeds every second [4]. Other examples include Netflix[1] and Amazon[2], with large collections of items and users. New ratings, items and users are being added to the ratings matrix constantly and need to be reflected in the top-k recommendations. Computing the list of top-k items repeatedly for all users or at login time for the active user can be costly, calling for efficient top-k algorithms.

One of the standard and most popular approaches for collaborative filtering is item-based collaborative filtering (CF). It is known to provide high quality top-k results [5,14]. The main idea behind this approach is to calculate a similarity matrix whose entries correspond to pairwise item similarities, Pearson Correlation [13] being one of the popular choices. Computing recommendations from scratch for active users can be prohibitively expensive so CF-based recommender systems typically precompute item-wise similarities and store them in a manner that facilitates quick retrieval [5]. Following this approach, the task of predicting the score of the active user (u_i) on an item (v_j) consists of the following steps:

- Find the N most similar items to v_j that u_i has rated, referred to as v_j's N *nearest neighbors* in u_i's *profile*.
- Compute a weighted average of u_i's ratings on the N items weighted by similarities, as the predicted rating or score of v_j by u_i.

Once item scores are predicted, those with highest predicted scores are served to the user. There has been a great

WOODSTOCK '97 El Paso, Texas USA

[1] http://www.netflix.com/
[2] http://www.amazon.com

deal of work on efficient algorithms for finding top-k items in various settings. In classic top-k settings, the scores of items on each of m features are available from m score-sorted lists. The challenge is to devise top-k algorithms that access the least number of items. This challenge is met by the TA and NRA family of algorithms [6] and their numerous descendants [3, 10, 17, 18, 21]. Compared with the classic settings, finding top-k in item-based CF raises *two unique major challenges*: (1) Unlike in previous work, we need to deal with the fact that the aggregate score of every candidate item must be determined based on entries from a *different set of lists*. This is because the N most similar items for every candidate item may be different in a given user profile. (2) Weighted average is *not a monotone aggregate function*, a property leveraged by classic top-k algorithms. While there has been some recent work on top-k algorithms with non-monotone aggregate functions [10, 18, 21], to our knowledge, no previous work on top-k addresses the combination of variable set of lists to aggregate from and non-monotone aggregate functions.

Our goal in this paper is to devise an approach for item-based CF[3] that is scalable while retaining the accuracy of known CF algorithms. We show that a direct approach based on extending TA/NRA-like ideas to the framework of CF either calls for unrealistic preprocessing and storage requirements or leads to algorithms which end up accessing as many entries as certain naive algorithms to be discussed later. Thus, we propose an alternative approach based on abstracting the key computation in CF by means of two operations that we call *probe* and *explore*. Probe is the process of identifying which (N) lists must be aggregated for a given candidate item and explore is the process of finding top-k items to be recommended, having found the lists to be aggregated. Our approach is based on using a similarity threshold value to make the probe step efficient. We propose a probabilistic cost-based approach whereby we can express the overall expected cost of our top-k algorithm as a function of the similarity threshold and determine the best value of the threshold that minimizes the cost.

We propose a scalable recommender system based on the core item-based CF method described above, with an acrhitecture consisting of the following layers (see Fig. 1): (i) Data layer, consisting of the user/item ratings matrix; (ii) Intermediate layer, consisting of the item-item similarity matrix, sufficient statistics and required routines for efficiently keeping the similarity matrix up-to-date. This layer also maintains the data structures used by top-k algorithms, and keeps them current; (iii) Application layer, including efficient top-k algorithms running on the top of the intermediate layer data. This layer invokes the probe operation using the similarity threshold determined through cost-based optimization, and finds top-k items via the explore operation.

Top-k algorithms can work independently from the intermediate layer and access up to date data structures from that layer. We explore the design space by considering two straightforward top-k algorithms and then propose a two-phase top-k algorithm. In the first phase, we calculate a threshold on similarity values. The similarity threshold is found in such a way as to maximize (resp., minimize) the probability that items with similarity values larger (smaller) then the threshold are among the N nearest neighbors of items. Efficiency and scalability come from the fact that after the first phase, as a result of using similarity threshold, a huge portion of the similarity matrix is filtered and the size of the problem is reduced significantly. The second phase performs a straightforward computation of predicted ratings using the surviving values. We define an objective function which is an upper bound on the expected cost of both phases put together and find the optimal value of the threshold that minimizes the cost. We model the similarity matrix with probability distribution(s) in order to facilitate probabilistic analysis. We propose an efficient algorithm for finding top-k recommendations. We make the following contributions:

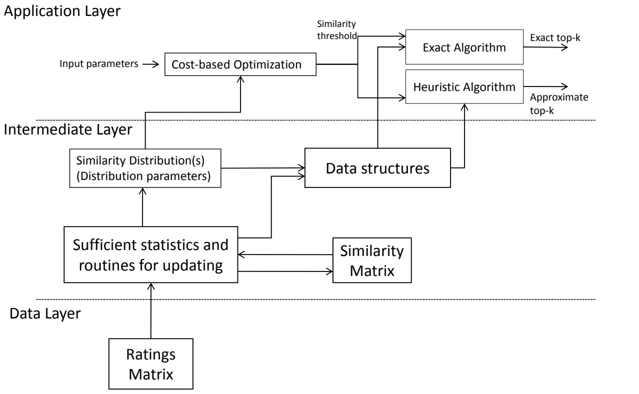


Figure 1: Layered architecture and its components

- We propose (Fig. 1) a layered system for item-based collaborative filtering. Our system is highly scalable and efficient while retaining the accuracy of CF algorithms. It also keeps the similarity matrix up-to-date with respect to new users and items added to data, as well as new ratings. This makes our system suitable for requirements of modern recommender systems.

- We propose two naive algorithms that serve as a baseline for calibrating the performance of the algorithms we consider. This naturally leads to a hybrid algorithm that combines their strengths (Sec. 3).

- We show that an approach based on extending TA/NRA style algorithms either requires an unreasonable amount of preprocessing and storage or leads to algorithms which can access as many entries as certain naive algorithms (Sec. 4).

- We design a novel top-k algorithm for item-based collaborative filtering. The algorithm is based on finding a similarity threshold value that optimizes the expected cost (Sec. 5).

- We evaluate the algorithms theoretically and empirically in terms of their running time and quality of top-k results. We empirically show our top-k algorithm is superior to the baseline methods in providing consistent high quality top-k recommendations in a more scalable way (Sec. 5 and Sec. 7).

Sec. 2 discusses the related work on scalable item-based CF methods and top-k algorithms. Sec. 8 concludes the paper.

[3] Henceforth, unless specified otherwise, by CF we mean item-based CF.

2. RELATED WORK

Item-based CF and Scalability: There has been extensive work on collaborative filtering. The prevalent approaches fall into the categories of user-based, item-based, model-based CF or fusion approaches. We refer the reader to [1,7,8] for excellent surveys. User-based methods predict unknown ratings based on ratings of N nearest users who have rated the item. This requires computing and maintaining an $n \times n$ user-user similarity matrix. The number of users n is typically much larger than the number of items m. Item-based CF was first proposed in [5, 14] to overcome these limitations. In addition to improving scalability, it has been found that pairwise item similarities are usually more reliable than pairwise user similarities and result in better accuracy [14]. In item-based CF, the rating of an active user on a candidate item is predicted based on her ratings on N items most similar to the candidate item. Sarvar et al. [14] propose a scalable approach which relies on storing only N (between 10 and 30) most similar items to every item rather than the whole matrix. A general drawback of item-based CF arises when the active users rated fewer than N items, resulting in loss of accuracy. In extreme cases, it may be impossible to make reasonable prediction of rating for some (user, item) combinations. A similar comment applies to user-based CF. Model-based methods [4] assume the ratings matrix has been generated from a model and use the observed data to learn optimal parameter values of the model. Despite their efficiency they suffer from lack of explainability. Besides, since they find model parameters through optimization, they have a training phase. Changing parameters incrementally and adapting to changes in data is not a trivial problem.

In sum, item-based CF is one of the popular approaches for collaborative filtering and devising scalable and efficient algorithms for its computation is of great importance. Indeed, there have been several works in the literature that aim at improving scalability of item-based methods [15, 19, 20]. In [15], singular value decomposition (SVD) is applied first in order to represent users and items in a lower dimensional space. Neighborhood formation is based on this compact representation of the original set of ratings. Our main focus is on top-k algorithms that run on top of the original and up to date similarity matrix. Another line of work has focused on instance selection for improving scalability [15, 19, 20]. Rather than considering the set of all users and items for finding nearest neighbors, they find a small subset that results in better accuracy compared to other subsets. Instance selection can indeed improve scalability albeit at the expense of accuracy. Besides, our approach is orthogonal to instance selection and one could use both instance selection together with our efficient top-k algorithms for improving scalability.

Top-K Algorithms: Top-k algorithms have been studied extensively. Most of the related work builds on top of classic TA and NRA algorithms [6]. These algorithms assume a monotone aggregate function and the set of lists from which the aggregation is performed is known beforehand. Compared to these algorithms, in our problem we face two major challenges: (i) our aggregation function is not monotone; (ii) the set of lists from which values are to be aggregated is not fixed and could change from item to item. Several recent works address the issue of non-monotone aggregation functions [10, 18, 21] although the issue of dealing with different sets of lists to aggregate scores from has not been addressed before, to our knowledge. The main idea leveraged in all cases is defining an upper bound on the score of unseen objects and a lower bound on the score of top-k items. The algorithm stops when the former becomes smaller than the latter. As we show in the paper, an adaptation of TA/NRA-like approaches to deal with changing set of lists for aggregating scores from leads to an algorithm which can be arbitrarily worse than the optimal algorithm on some instances. In contrast, we propose a significantly different algorithm for computing top-k items. It is based on organizing top-k computation in terms of probes and explores and choosing an optimal similarity threshold using cost-based optimization. We propose a scalable exact algorithm and an even faster approximate algorithm.

3. PRELIMINARIES AND NAIVE ALGORITHMS

In collaborative filtering, data is represented as a sparse $n \times m$ matrix R, with entry r_{ij}, the rating of i^{th} user on j^{th} item, taking on integer values from the set $\{1, 2, ..., C\}$, for some $C > 1$.

The main challenge is to predict the missing ratings in R. Henceforth, we use r_{ij} to refer to the *existing ratings* of i^{th} user on j^{th} item. We refer to i^{th} user (row) as u_i and to j^{th} item (column) as v_j. Notice that from a behavioral perspective, we can identify the i^{th} user (resp., j^{th} item) with row u_i (resp., column v_j). When we refer to u_i (resp., v_j) below, we mean either user i (resp., item j) or row i (resp., column j). We use $r \in u_i$ (resp., $r \in v_j$) to denote an *existing rating* in user u_i's row (item v_j's column). We use $\widehat{r}_{ij}$ for the predicted value of the active user u_i's unknown rating on a candidate item v_j. In the rest of this paper we use μ_i to refer to the number of items rated by u_i and μ to denote the average number of items rated by any user.

Prediction of $\widehat{r}_{ij}$ in item-based collaborative filtering is done typically by taking the weighted average of ratings of u_i on N most similar items to v_j. More formally, we can use the following equation, where $\mathbf{N}(v_j, u_i)$ denotes the set of N items that are most similar to v_j and are rated by u_i:

$$\widehat{r}_{ij} = (\sum_{x=1}^{N} s_{xj} \times r_{ix}) \,/\, (\sum_{v_y \in \mathbf{N}(v_j,u_i)} s_{yj}) \qquad (1)$$

Problem Studied: Given an active user u_i, the problem is to efficiently find the top-k items with the highest predicted rating. We call this the problem of finding top-k recommendations. Predicting the rating of a candidate item v_j requires identifying the N items that are most similar to v_j and are rated by u_i, and then aggregating their ratings using Eq. 1.

The first natural question, given the body of work on the TA/NRA family of top-k algorithms, is whether we can adapt those algorithms and design efficient top-k algorithms for finding top-k recommendations. We address this question in Section 4. In the next subsections, we present two naive algorithms that will serve as baselines for the top-k algorithms considered in this paper. For convenience, for a given item v_j, by the N nearest neighbors of v_j we mean the N most similar items to v_j.

3.1 Algorithm Naive-1

For both Algorithms Naive-1 and Naive-2, we assume pairwise item similarities are materialized. A straightforward procedure for finding top-k items with respect to u_i is as follows:

1. Predict the score[4] of each individual candidate item, v_j, using Eq. 1. This involves finding N most similar items to v_j in u_i's profile, i.e., the set $\mathbf{N}(v_j, u_i)$. Finding K items with highest values in a list of m items has been studied extensively in the literature as the *K-Select* problem [9]. Here, since values of N are in the range 10-30 in practice, we can perform this task by a simple scan of the list and maintain a priority queue. This is doable in $O(mlogN)$. Since N is typically a small number, this approach is typically more efficient than the best deterministic algorithm for the general *K-Select* problem which takes $\Theta(10m)$ using the median of medians approach [9] and we prefer to use it.

   After finding N nearest neighbors of each item (rated by u_i), aggregating their ratings and calculating $\widehat{r}_{ij}$ takes $O(N)$ time. Thus, the total cost of predicting the score of an individual candidate item is $O(mlogN + N)$ and the cost of predicting scores of all candidate items is $O(m^2 logN + mN)$. In practice, it is possible to improve this by searching for N nearest neighbors of v_j only within the list of items rated by the user, which will result in $O(m\mu_i logN + mN)$ cost where μ_i is the number of items rated by user u_i. This improvement is possible provided the list of items rated by any given user is stored separately in addition to the ratings matrix. Storing such information adds a cost of $O(n\mu)$ to the storage, where μ is the average number of items rated by a user. This is of the same order as the cost of efficiently storing the sparse ratings matrix and we can assume access to this information.

2. Once the score of every item is calculated in previous step, find k items with highest scores in $O(mlogk)$.

The total running time of the above algorithm is $O([\mu_i logN + N + logk] \times m)$.

We can see that finding the top-k items to recommend to a given user involves two steps: (i) ***Probe*** every candidate item in order to find the N items in the user's profile that are most similar to it. (ii) ***Explore*** the list of candidate items and calculate their scores and then find the top-k items that have the highest scores. Step (ii) is necessary in order to find the best scoring k items. For any item, determining its score requires that we know the N most similar items to it, which is addressed by step (i).

The probe step involves $O(m\mu_i logN)$ operations while explore requires $O(m(logk + N))$ operations. In practice, values of N are typically 30 or smaller. Also, $logk$ does not typically exceed 5, the main reason being we don't want to overwhelm users with too many recommendations[5]. Existence of a μ_i factor in the probe cost suggests that the probe component in the cost of Algorithm Naive-1 dominates the explore cost. To put this in perspective, for instance, in the MovieLens dataset [11] with 1 Million ratings the average value of μ_i is around 165. Also in Netflix [12] dataset with $100M$ ratings and $500k$ users, the average number of ratings per user is 200.

3.2 Algorithm Naive-2

We define a data structure, L, besides similarity matrix which is a collection of sorted lists, one for each item. A schematic of L is illustrated in Figure 2. Denote by $L_i \in L$ the list associated with item v_i. Elements of L_i are pairs of the form $(item\_pointer, similarity)$ and lists are sorted in non-increasing similarity w.r.t. v_i; *item_pointer* is a pointer to the actual memory location where item resides or simply the item id. Pointers are used to have a unified representation of items between the ratings matrix R and the similarity lists L. We use L_{ij} to denote the j^{th} entry of list L_i. $L_{ij}.similarity$ denotes the similarity of j^{th} most similar item to v_i. Each of the sorted lists in L is maintained as a priority queue to facilitate efficient updates as item similarities change over time. Algorithm Naive-2 works as follows:

1. Mark all μ_i items rated by u_i which takes time $O(\mu_i)$.

2. For every candidate item v_j, read L_j from the head of the list until N marked items are found.

3. Calculate the aggregate scores and find k items with highest scores in time $O(mlogk)$.

There is a total of m items out of which μ_i are marked. Thus, a randomly chosen item has equal probability of being at any position of the list. Therefore, we can assume μ_i marked items divide the list into $\mu_i + 1$ buckets. The expected number of unmarked items in each bucket is $(m - \mu_i)/(\mu_i + 1)$ and the expected number of items to visit until N marked items are observed is $N + N \times \frac{(m-\mu_i)}{(\mu_i+1)}$ or $O(Nm/\mu_i)$ which makes the expected running time of algorithm $O(mN(m/\mu_i) + mlogk)$.

It is easy to verify Algorithm Naive-2 is expected to be more efficient than Algorithm Naive-1 when $\mu_i > \sqrt{Nm/logN}$. Combining the two algorithms, using Naive-1 when $\mu_i \leq \sqrt{Nm/logN}$ and Naive-2 when $\mu_i > \sqrt{Nm/logN}$, results in a procedure with expected running time $O(mlogk + m \times min(\mu_i logN, Nm/\mu_i))$. We call this Algorithm Hybrid. A point worth making is that the theoretical "switchover" value of $\mu_i > \sqrt{Nm/logN}$ was obtained based on the worst case expected cost of the Algorithms Naive-1 and Naive-2. In our experiments, we empirically explore the best switchover value on the Netflix data set.

4. CLASSIC TOP-K ALGORITHMS

Since TA/NRA style algorithms rely on the aggregation function being monotone, a straightforward approach for leveraging those algorithms is to turn the problem into one where the aggregate function is monotone. In Eq. 1, define the contribution of item v_x rated by user u_i to the predicted rating of item v_j as $e_j^x = s_{xj} r_{ix} / \sum_{v_y \in \mathbf{N}(v_j, u_i)} s_{yj}$. It is easy to see the predicted rating is $\widehat{r}_{ij} = \sum_{v_x \in \mathbf{N}(v_j, u_i)} e_j^x$. The TA/NRA family of algorithms can now be applied since *sum* is a monotone aggregation function. However, in order to realize this, we need to maintain N lists storing similarities of each candidate item v_j to each of its N nearest neighbors that are rated by u_i, for each user u_i. This requires finding the N most similar items for every candidate item that are rated by each user and it requires an $O(nmN)$ storage, which is prohibitive. Besides storage, it requires keeping the list of N nearest neighbors of every item up to date for every user which is also computationally expensive. In practice,

[4] We use score and rating interchangeably.

[5] Netflix (www.netflix.com) recommends a list of 25 items to users

we only need to find top-k recommendations for every user when they enter the system.

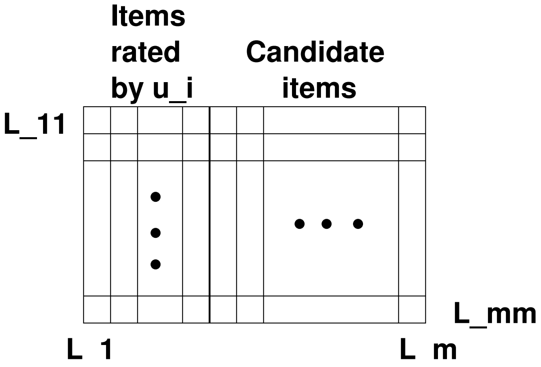


Figure 2: Schematic of data structure L storing item-wise similarities in non-increasing order. We assume active user u_i has rated first μ_i items.

In this paper, we seek efficient top-k algorithms with efficient storage requirements. More specifically, the minimum storage requirement is the ratings matrix and the materialized pairwise item similarities. Since we do not want to store this information on a per user basis, we need to materialize all pairwise similarities in general. A data structure consisting of lists corresponding to every item, storing similarities to other items in non-increasing order, is an efficient implementation of the similarity information and we will assume the top-k algorithms have access to this information. This data structure, called L, is schematically depicted in Fig. 2 where we assume active user u_i has rated the first μ_i items. We discuss two different possible realizations of TA/NRA style algorithms using the above idea and show there are instances where adaptations of TA/NRA algorithms will access all the entries accessed by one of the naive algorithms discussed earlier. This will set the stage for developing more efficient algorithms.

More specifically, we consider algorithms that access entries in columns of L in sorted similarity order. Once they read a similarity value between a candidate item and one of the μ_i items rated by the user, they wait until they can decide whether the rated item is among the N nearest neighbors of candidate item or not.[6] After finding a new neighbor for a candidate item, they update the necessary upper and lower bounds. Finally they terminate once the lower bound on the score of top-k items becomes no less than the upper bound on the score of partially observed objects. In the rest, we call such algorithms *"classic algorithms"*. We consider two types of classic algorithms: (i) those which access columns of L corresponding to items rated by u_i in sorted order; (ii) those which access columns of L corresponding to candidate items. We make no assumptions about the order on which lists are chosen for probing and the depth until which each of them is probed. We prove in both cases for any classic algorithm as mentioned above, there exists an instance on which the algorithm will access as many instances as one of the naive algorithms.

THEOREM 1. *Let CA be any classic algorithm that reads similarity values from those columns of L that correspond to items rated by u_i. There is an instance on which CA will access as many entries as Algorithm Naive-1.*

PROOF. Consider an instance where there are N items rated C (maximum score) by user u_i. Call them (u_i's) favorite items. Assume there is at least one item among the top-k whose aggregate score is less than C. Let $S_{min} = min(\{L_{jm}.similarity|v_j \in u_i\})$ which is the minimum similarity value in the bottom row of L, among columns corresponding to items rated by u_i. Choose S_1 and S_2 such that $S_2 < S_1 < S_{min}$. Add a new candidate item to L, v_{new}, such that the last entry of every list corresponds to v_{new}. Set similarities of v_{new} with the N favorite items to S_1, and with the rest of items to S_2. We will show any classic algorithm needs to access all of the entries of the μ_i columns in order to find the top-$(k+1)$ items.

Any classic algorithm will access entries in the μ_i columns in sorted order. Finding the top-$(k+1)$ items is not possible unless all the N nearest neighbors of v_{new} are found. This is because: (i) v_{new} is among the top-$(k+1)$ items, since its aggregated score is C; (ii) The lower bound on the score of the top-$(k+1)$ items remains less than the upper bound on the the score of partially observed items until all the N nearest neighbors of v_{new} are seen. This is because there is at least one item among the top-$(k+1)$, whose score is less than maximum possible score C.

Finding all of the N nearest neighbors of v_{new} requires reading all of the entries at the bottom of the μ_i lists. This is due to the fact that the maximum similarity of v_{new} to items rated by u_i is less than S_{min}. Therefore, CA can't use other similarity values in order to find N nearest neighbors of v_{new}. But then reading all of the similarities between v_{new} and the μ_i items rated by the user forces CA to read all of the entries in all the μ_i lists since it reads entries of L in sorted order. □

THEOREM 2. *Let CA be any classic algorithm that reads similarity values from those columns of L that correspond to candidate items. There is an instance on which CA will access as many entries as Algorithm Naive-2.*

PROOF. Consider an instance where there are no items rated C (maximum score) by user u_i. Assume the instance is such that there are no two candidate items whose N nearest neighbors are exactly the same. Consider the sequence of similarity values accessed by CA and let's call it Seq_s. Notice that the order in which similarity values appear in Seq_s does not depend on ratings of u_i since these are similarities w.r.t. candidate items (not rated by u_i). It only depends on the order in which candidate lists are probed and until which depth. Also note that no list will be probed beyond the point where the N^{th} most similar item is observed. CA will terminate at some point when the lower bound on the score of top-k items becomes no less than the upper bound on the score of partially observed objects. Assume this termination criterion is removed from the algorithm and Seq_s is continued until the last N^{th} nearest neighbor is observed. Call the item whose N^{th} nearest neighbor is observed last v_{last}. We create a new instance by changing ratings of u_i on N nearest neighbors of v_{last} to C. Obviously, v_{last} will be the top item in the new instance. Now, the upper bound on the score of partially observed items will remain greater than the lower bound on the score of top-k items, unless the N nearest neighbors of v_{last} (and thus of all items) are ac-

[6] In some situations decision can be made immediately

cessed. Therefore any classic algorithm will access as many entries of L as Algorithm Naive-2. □

The above two theorems highlight the fact that in theory the performance of classic algorithms can be as bad as the two naive algorithms in terms of the number of entries of L they access. It is worth noting however that an efficient implementation of the classic algorithms needs to maintain the lower and upper bounds, which requires careful bookkeeping and may lead to additional storage and computational cost. Besides, getting good lower and upper bounds with non-monotone aggregate functions is an additional challenge. Motivated by these issues, we propose an alternative two-phase approach to the top-k recommendations problem based the operations probe and explore. The details are described in the following section. As a preview, we will show (Sec. 5.2) that for various choices of values for the parameters μ_i and N, the expected cost of our two-phase algorithm in terms of entries accessed rarely exceeds that of the two naive algorithms and is most of the time much less. A point worth noting is unlike our two-phase algorithm, the performance of classic algorithms cannot be predicted based on the values of these parameters alone, since for any choice, we can always find an instance where the classic algorithms access as many entries as one or the other naive algorithm.

5. A TWO-PHASE ALGORITHM

Recall, between the steps probe and explore (Sec. 3), probe is much more expensive. To facilitate efficient implementation of probe, in this section, we describe a two-phase process which is the basis for our efficient top-k algorithm.

5.1 Two-Phase Algorithm

We start with the data structure L described in Sec. 3 and extend it by adding a third element to every entry. We call the new data structure LP. Specifically, in LP, an entry LP_{ij} is a triple where $LP_{ij}.item\_pointer$ and $LP_{ij}.similarity$ are the same as the corresponding elements in L. The third element $LP_{ij}.prob$, represents the probability that any other similarity value in L is larger than $LP_{ij}.similarity$. Although all of the similarities are available in the similarity matrix, reading all of the relevant similarities is costly. We use these probabilities to make probabilistic decisions during execution. Making such decisions, we seek to avoid accessing all similarity values for rated items (unlike Algorithm Naive-1) or accessing all similarity values of candidate items until, for each of them, the N nearest neighbors are found (unlike Algorithm Naive-2). So we first read only values that have a high probability of being among the N most similar items of the candidate items. We do this by finding a threshold and filtering all entries with prob values above it. Then, we search for N nearest neighbors of candidate items among the surviving values. It is possible that for some items we have missed some or all of the nearest neighbors. In such cases, we probe for the remaining neighbors only for those items. Obviously, choosing the right threshold value is of great importance. We do this through cost based optimization.

$LP_{ij}.prob$ values are estimated through modeling of similarity matrix with an appropriate distribution, the details of which we will explain later. As we will show later, when we model similarity matrix with a distribution, *there is no need to materialize prob values.* Instead, the probability threshold can be directly translated into a similarity threshold. For simplicity of presentation, here let us assume these probabilities are given. Notice that elements of LP that contain higher similarity values, have lower *prob* values. Therefore, keeping columns of LP sorted in non-increasing order of similarities is identical to keeping them sorted in non-decreasing order of *prob* values.

Fig. 3 illustrates the two-phase process as well as naive algorithms for an active user (u_i), who has rated 3 out of 7 items. In Fig. 3, on the left side of the top row, the three columns of LP corresponding to items rated by u_i are shown and all entries with *prob* values below an example threshold value of $\theta = 0.19$ are highlighted. On the right side, inverted lists of candidate items are shown. These lists contain only the surviving similarity values. Assuming $N = 2$, we are done with finding neighbors of v_4 and v_7. We have found the nearest neighbor of v_5 as well and only need to find the second nearest neighbor among the remaining items rated by u_i (i.e., v_1 and v_2). This reduces the execution time for three out of four candidate items compared to Algorithm Naive-1, which needs to search for the nearest neighbors of *all items* among all μ_i items rated by u_i (see Fig. 3, bottom right). However, note that the two-phase approach still needs to pay the cost of reading some irrelevant similarity values such as s_{13}. In addition, finding the remaining neighbors of v_5 requires at least a scan of the μ_i items rated by the user. All of this underscores the importance of choosing the right threshold for purpose of pruning. We do this by cost-based optimization.

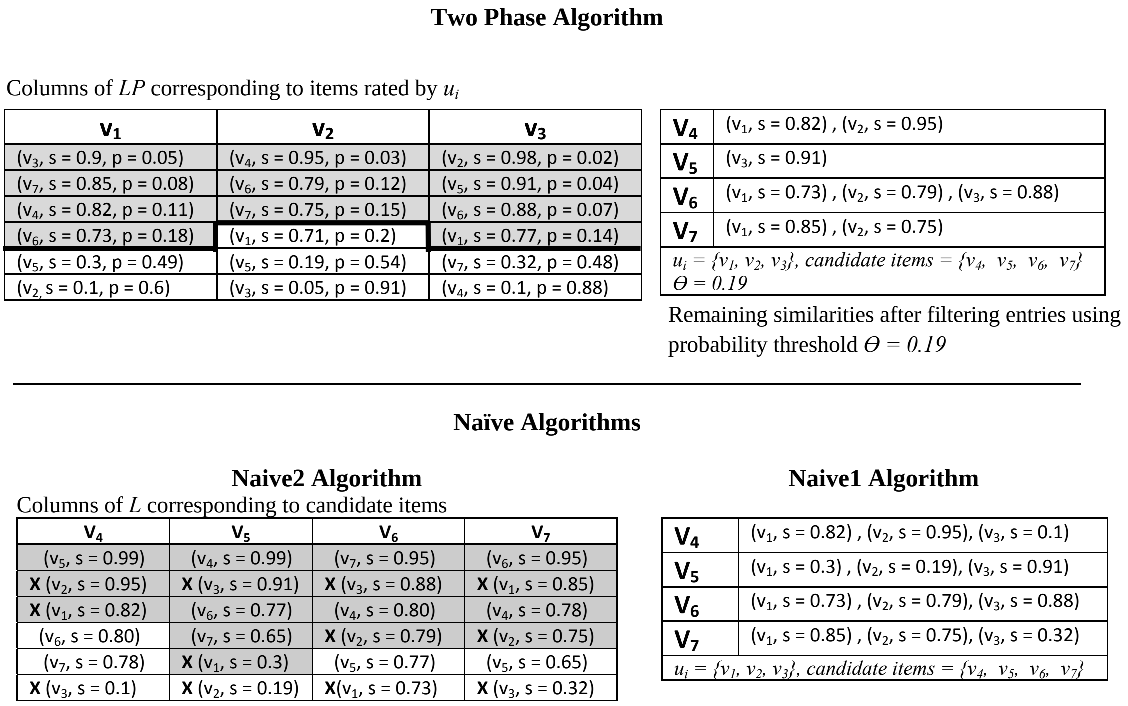
Fig. 3 (bottom left) also illustrates how Algorithm Naive-2 works. In this specific example, almost half of the items are rated and the scenario is in favor of this algorithm.[7] However, the two-phase process accesses only 11 entries as opposed to 16 by Naive-2 and 12 by Naive-1. Our purpose in this example is just to illustrate how the two-phase process works and the need for cost-based optimization in choosing the probabilistic threshold. Toward the end of Sec. 5.2, we compare the expected cost of the two-phase algorithm with the cost of Algorithms Naive-1 and Naive-2.

Following the two phases of probe, the explore step is applied in order to find the top-k items.

In order to compare two phase process to classic top-k algorithms, in Fig. 3, assume v_5 is one of the top-k items. Also assume u_i has provided the maximum rating (C) to both of the nearest neighbors of v_5. In this case, the upper bound on the score of partially observed items (in the sense of the classic top-k algorithms) won't drop until we have found both of the nearest neighbors of v_5. Furthermore, we need to make sure there is no other similarity value between v_5 and other items rated by the user which are larger than the ones we have read. It is easy to verify that doing so either requires reading 5 more entries or it requires reading 3 more entries and performing rigorous book keeping to keep track of under which lists we have observed every item, an expensive step in itself. In the worst case, this could be as bad as reading all entries which is even worse than Algorithm Naive-1. Therefore, we propose an alternative approach based on the two-phase process and choose the threshold based on cost-based optimization.

5.2 Choosing Threshold Prob Value θ

[7] In practice, e.g., in Netflix, the number of items rated by a user on an average is a small fraction of the number of all items, which makes Algorithm Naive-2 perform even worse.



Two Phase Algorithm

Columns of *LP* corresponding to items rated by u_i

v_1	v_2	v_3
(v_3, s = 0.9, p = 0.05)	(v_4, s = 0.95, p = 0.03)	(v_2, s = 0.98, p = 0.02)
(v_7, s = 0.85, p = 0.08)	(v_6, s = 0.79, p = 0.12)	(v_5, s = 0.91, p = 0.04)
(v_4, s = 0.82, p = 0.11)	(v_7, s = 0.75, p = 0.15)	(v_6, s = 0.88, p = 0.07)
(v_6, s = 0.73, p = 0.18)	(v_1, s = 0.71, p = 0.2)	(v_1, s = 0.77, p = 0.14)
(v_5, s = 0.3, p = 0.49)	(v_5, s = 0.19, p = 0.54)	(v_7, s = 0.32, p = 0.48)
(v_2, s = 0.1, p = 0.6)	(v_3, s = 0.05, p = 0.91)	(v_4, s = 0.1, p = 0.88)

V_4	(v_1, s = 0.82) , (v_2, s = 0.95)
V_5	(v_3, s = 0.91)
V_6	(v_1, s = 0.73) , (v_2, s = 0.79) , (v_3, s = 0.88)
V_7	(v_1, s = 0.85) , (v_2, s = 0.75)
$u_i = \{v_1, v_2, v_3\}$, *candidate items* $= \{v_4, v_5, v_6, v_7\}$ $\Theta = 0.19$	

Remaining similarities after filtering entries using probability threshold $\Theta = 0.19$

Naïve Algorithms

Naive2 Algorithm

Columns of *L* corresponding to candidate items

V_4	V_5	V_6	V_7
(v_5, s = 0.99)	(v_4, s = 0.99)	(v_7, s = 0.95)	(v_6, s = 0.95)
X (v_2, s = 0.95)	**X** (v_3, s = 0.91)	**X** (v_3, s = 0.88)	**X** (v_1, s = 0.85)
X (v_1, s = 0.82)	(v_6, s = 0.77)	(v_4, s = 0.80)	(v_4, s = 0.78)
(v_6, s = 0.80)	(v_7, s = 0.65)	**X** (v_2, s = 0.79)	**X** (v_2, s = 0.75)
(v_7, s = 0.78)	**X** (v_1, s = 0.3)	(v_5, s = 0.77)	(v_5, s = 0.65)
X (v_3, s = 0.1)	**X** (v_2, s = 0.19)	**X**(v_1, s = 0.73)	**X** (v_3, s = 0.32)

Naive1 Algorithm

V_4	(v_1, s = 0.82) , (v_2, s = 0.95), (v_3, s = 0.1)
V_5	(v_1, s = 0.3) , (v_2, s = 0.19), (v_3, s = 0.91)
V_6	(v_1, s = 0.73) , (v_2, s = 0.79), (v_3, s = 0.88)
V_7	(v_1, s = 0.85) , (v_2, s = 0.75), (v_3, s = 0.32)
$u_i = \{v_1, v_2, v_3\}$, *candidate items* $= \{v_4, v_5, v_6, v_7\}$	

Figure 3: An example of running the two phase probe step using a prob threshold of $\theta = 0.19$ and comparing it to naive algorithms

We already illustrated with an example how given a threshold on *prob* values, the probe step is performed in two phases. We choose this threshold value in such a way as to minimize an upper bound on the expected overall cost of the algorithm. In this section we introduce the necessary functions that serve as components of the cost function.

DEFINITION 1. *Let v be an item in a user's profile, v_c be a candidate item, s be the similarity of v w.r.t. v_c, and p be the corresponding prob value. We use $Q(p)$ to denote the probability that v is among the N nearest neighbors of v_c.*

Assume μ_i is the number of items rated by user u_i. Let v_j be one of the items rated by u_i and let v_c be a candidate item (and thus unrated by u_i) with similarity s to item v_j. Let p be the corresponding *prob* value, i.e., the probability that any other given similarity value in LP is larger than s. We define a Bernoulli random variable X_ℓ, where $X_\ell = 1$ if v_j is the ℓ^{th} most similar item to v_c among items rated by u_i and $X_\ell = 0$ otherwise. We have:

$$P(X_\ell = 1) = \begin{pmatrix} \mu_i - 1 \\ \ell - 1 \end{pmatrix} p^{\ell-1}(1-p)^{\mu_i-\ell} \tag{2}$$

In Equation 2, $P(X_\ell = 1)$ is equivalent to $\ell - 1$ successful trials out of $\mu_i - 1$ where p is the probability of success, i.e., that some other similarity value in LP is higher than s. Now let's define another random variable, Y, which takes on value 1 if v_j is one of the N nearest neighbors of v_c among items rated by u_i and 0 otherwise. Equation 3 defines $Q(p)$ in terms of Y:

$$\begin{aligned} & Q(p) = P(Y = 1) \\ & = \sum_{\ell=1}^{N} P(X_\ell = 1) \\ & = \sum_{\ell=0}^{N-1} \begin{pmatrix} \mu_i - 1 \\ \ell \end{pmatrix} p^{\ell}(1-p)^{\mu-\ell-1} \end{aligned} \tag{3}$$

DEFINITION 2. *For a given user u_i, define the* core set *of u_i to consist of all of the similarity entries that contribute to the scores of all candidate items, i.e., similarity values between all candidate items and their N nearest neighbors in u_i's profile.*[8]

Assuming there are m items and $m >> \mu_i$ for any given u_i, there are $O(m)$ candidate items. Thus the core set consists of $O(Nm)$ similarity values. For any given entry l of LP, $P(l.similarity \in Core) = Q(l.prob)$. This means in order for a similarity value to be in the core set, it has to be among the N nearest neighbors of the candidate item corresponding to $l.item\_pointer$.

DEFINITION 3. *We use S_θ to denote the set of surviving similarity values in phase 1, after filtering out all entries from LP whose prob values are no less than the threshold value θ, as described earlier and illustrated with Fig. 3.*

Notice, $Core - S_{\theta_a}$ is the set of all similarity values in the core set that are not obtained in phase 1 and hence would

[8] Note, two different pairs of items having the same exact similarity value, are counted as two different member entries of the core set.

need to be processed in the second phase. We have the following result, establishing an upper bound on the expected size of this set.

PROPOSITION 1. *For any threshold value θ_a, $Q(\theta_a)m\mu_i$ is an upper bound on the expected size of $|Core - S_{\theta_a}|$.*

PROOF. Let $A = \{LP_{j\kappa} \mid LP_{j\kappa}.prob > \theta_a$ and $v_j \in u_i\}$. Notice, $\forall l \in A$, $Q(l.p) < Q(\theta_a)$. We already know $P(l.s \in Core) = Q(l.p)$.
$\rightarrow E(|A \cap Core|) = |A| \times Q(l.p)$
$\rightarrow E(|A \cap Core|) < Q(\theta_a)m\mu_i$. Since $|A \cap Core| = |Core - S_{\theta_a}|$, the proof is complete. □

Given a threshold value θ_a, in the first phase all entries with $prob$ value below θ_a are read resulting in an expected number of at most $m\mu_i\theta_a$ values that are carried forward to the next phase. This is due to the fact that the expected number of entries with pointers to candidate item v_c, are $\mu_i\theta_a$ and there are at most m candidate items. Finding N highest values in list of length $\mu_i\theta_a$ results in the expected cost of $\mu_i log(N)\theta_a$ for a single candidate item. We know the expected number of missing entries from $Core$ in S_{θ_a} is at most $Q(\theta_a)m\mu_i$. Here, we consider a worst case scenario. Let's assume all of the missing entries belong to different items. Also assume for all of these items we perform the task of finding N nearest neighbors from scratch on a list of μ_i items. Therefore, an upper bound on the expected number of items for which the probe step needs to be done from scratch is $Q(\theta_a)m\mu_i$.[9] We also know that $m\mu_i\theta_a$ is an upper bound on the number of irrelevant similarity values (similarities between items rated by the user). Taking all these factors into account, an upper bound on the expected overall cost of both phases, given a threshold value θ_a on $prob$ values is provided in Equation 4.

$$\begin{aligned} &C(\theta_a) = Q(\theta_a)m{\mu_i}^2 \log(N) \\ &+(m - Q(\theta_a)m\mu_i) \log(N)\mu_i\theta_a \\ &+m\mu_i\theta_a \\ &\overset{m\mu_i\times}{=} Q(\theta_a)\mu_i \log(N)(1-\theta_a) + \theta_a(1+\log(N)) \end{aligned} \tag{4}$$

Since the variable of the function is θ_a, we drop the $m\mu_i$ which is factored out in the rest. The optimal value of threshold (θ) given the cost function can be found as follows:

$$\theta = \arg\min_{\theta_a}(C(\theta_a)) \tag{5}$$

As can be seen in Equation 4, there are two competing components. On one hand, the expected number of missing entries from $Core$ set decreases if we use a larger threshold. On the other hand, using a large threshold results in a weak pruning in the first phase and it won't be effective enough to improve the efficiency of the second phase. Finding optimal threshold value requires optimization of the cost function. We will first show the cost function in Equation 4, has only one minimum as long as $\mu_i \geq 2$ and $N \geq 2$, and then explain how to perform the optimization. Before that, we need the following two important lemmas.

LEMMA 1. *The cost function $C(\theta_a)$ has at least one minimum as long as $0 < \theta_a < 1$, $\mu_i \geq 2$ and $N \geq 2$.*

PROOF. Equation 6, provides derivative of the cost function after dropping the $m\mu_i$ factor.

$$\begin{aligned} &\tfrac{\partial C(\theta_a)}{\partial \theta_a} = \tfrac{\partial Q(\theta_a)}{\partial \theta_a}\mu_i \log(N)(1-\theta_a) \\ &-Q(\theta_a)\mu_i \log(N) + (1+\log(N)) \end{aligned} \tag{6}$$

We need to take derivative of the Q function with respect to θ_a as well.

$$\begin{aligned} &\tfrac{\partial Q}{\partial \theta} = -(\mu_i - 1)(1-\theta)^{\mu_i - 2} \\ &+[\sum_{\ell=1}^{N-1} \begin{pmatrix} \mu_i - 1 \\ \ell \end{pmatrix} \ell \times \theta^{\ell-1}(1-\theta)^{\mu_i-\ell-1} \\ &- \begin{pmatrix} \mu_i - 1 \\ \ell \end{pmatrix} (\mu_i - \ell - 1) \times \theta^{\ell}(1-\theta)^{\mu_i-\ell-2}] \end{aligned}$$

While

$$\begin{aligned} &\sum_{\ell=\kappa}^{\kappa+1}[\begin{pmatrix} \mu_i - 1 \\ \ell \end{pmatrix} \ell \times \theta^{\ell-1}(1-\theta)^{\mu_i-\ell-1} \\ &- \begin{pmatrix} \mu_i - 1 \\ \ell \end{pmatrix} (\mu_i - \ell - 1) \times \theta^{\ell}(1-\theta)^{\mu_i-\ell-2}] \\ &= \begin{pmatrix} \mu_i - 1 \\ \kappa \end{pmatrix} \kappa \times \theta^{\kappa-1}(1-\theta)^{\mu_i-\kappa-1} \\ &- \begin{pmatrix} \mu_i - 1 \\ \kappa+1 \end{pmatrix} (\mu_i - \kappa - 2) \times \theta^{\kappa+1}(1-\theta)^{\mu_i-\kappa-3} \end{aligned}$$

This is because the the second term inside the sum when $\ell = \kappa$, is canceled with the first term when $\ell = \kappa + 1$. Altogether, we can simplify the sum and represent the derivative of Q in Equation 7.

$$\frac{\partial Q(\theta_a)}{\partial \theta_a} = -\begin{pmatrix} \mu_i - 1 \\ N-1 \end{pmatrix} (\mu_i - N)\theta_a^{N-1}(1-\theta_a)^{\mu_i-N-1} \tag{7}$$

Now, it could be verified easily that $C(\theta_a)$ is a degree μ_i polynomial and therefore continuous for $0 < \theta_a < 1$. Also

$$\lim_{\theta_a \to 0}(C'(\theta_a)) = -\mu_i \log(N) + \log(N) + 1$$

Which is negative when $\mu_i \geq 2$ and $N \geq 2$. And,

$$\lim_{\theta_a \to 1}(C'(\theta_a)) = \log(N) + 1$$

Since the sign of derivative changes from negative to positive at least once we conclude that Cost function ($C(\theta)$) has at least one minimum. □

In the expression for $C'(\theta_a)$, the derivative of $C(\theta_a)$ w.r.t. θ_a, the sub-expression that depends on θ_a is given by:

$$B(\theta_a) = \log(N)\mu_i[Q'(\theta_a)(1-\theta_a) - Q(\theta_a)] \tag{8}$$

where $Q'(\theta_a)$ is the derivative of $Q(\theta_a)$. We have:

LEMMA 2. *The function $B(\theta_a)$ has only one minimum if $0 < \theta_a < 1$.*

[9] We consider this worst case scenario only in our optimization. Our implementation takes advantage of the partial list of neighbors already found and performs the task more efficiently.

PROOF.

$$\begin{aligned}
&B(\theta_a) = \log(N)\mu_i \\
&\times[-\binom{\mu_i - 1}{N-1}(\mu_i - N){\theta_a}^{N-1}(1-\theta_a)^{\mu_i - N} \\
&-Q(\theta_a)] \rightarrow
\end{aligned}$$

$$\begin{aligned}
&B'(\theta_a) = \log(N)\mu_i \times (\\
&-\binom{\mu_i - 1}{N-1}(\mu_i - N) \\
&\times[(N-1){\theta_a}^{N-2}(1-\theta_a)^{\mu_i - N} \\
&-(\mu_i - N){\theta_a}^{N-1}(1-\theta_a)^{\mu_i - N-1}] \\
&-\binom{\mu_i - 1}{N-1}(\mu_i - N){\theta_a}^{N-1}(1-\theta)^{\mu_i - N-1})
\end{aligned}$$

We set $B'(\theta_a) = 0 \rightarrow$

$$ {\theta_a}^{N-2}(1-\theta_a)^{\mu_i - N-1}(N - 1 - (\mu_i - 2)\theta_a) = 0$$

The only value between 0 and 1 that can make $B'(\theta_a) = 0$ is $\theta_a = (N-1)/(\mu_i - 2)$. It could be verified that the second derivative of B at this point is positive and therefore $B(\theta_a)$ has only one minimum in the range. □

We now have:

THEOREM 3. *The cost function $C(\theta_a)$ given in Equation 4 has only one minimum under the following assumptions*

$$0 < \theta_a < 1, \mu_i \geq 2, N \geq 2.$$

PROOF. Using Lemma 2, we know that $C'(\theta_a)$ has only one minimum. Let's call the value of θ_a at this point θ_{min}. This means $C'(\theta_a)$ is monotonically decreasing for $0 < \theta_a < \theta_{min}$ and monotonically increasing for $\theta_{min} \leq \theta_a < 1$. Therefore, there are at most two possible points at which $C'(\theta_a)$ could cross the x-axis. We know

$$\lim_{\theta_a \rightarrow 0}(C'(\theta_a)) = -\mu_i \log(N) + \log(N) + 1$$

which is negative given $\mu_i \geq 2$ and $N \geq 2$. Since the value of $C'(\theta_a)$ is monotonically decreasing until θ_{min}, it can't cross the x-axis in this range. Therefore, given the fact that $C'(\theta_a)$ is monotonically increasing after θ_{min}, there could be at most one point at which it crosses the x-axis. Using Lemma 1 we know $C(\theta_a)$ has at least one minimum. Therefore, the function $C(\theta_a)$ has exactly one minimum under the given assumptions. □

Notice the assumptions are quite reasonable in practice. Finding the only zero of $C'(\theta)$ in closed form is intractable since it's a high degree polynomial. Instead, we use a numerical approach for doing this. In practice it takes only a few milliseconds to closely approximate the only zero of $C'(\theta_a)$ using binary search. A few milliseconds are quite affordable since we found the amount of time required to provide top-k items is typically less than the delay before a web page is loaded.

Using the optimal threshold value obtained from our optimization step, we compare the theoretical upper bound on expected cost of the two-phase algorithm with cost of Algorithm Naive-1 and the expected cost of Algorithm Naive-2, for different values of μ_i and N. Since Naive-1 needs to process all of the μ_i items rated by the user, its exact cost can be calculated as described in Sec. 3.

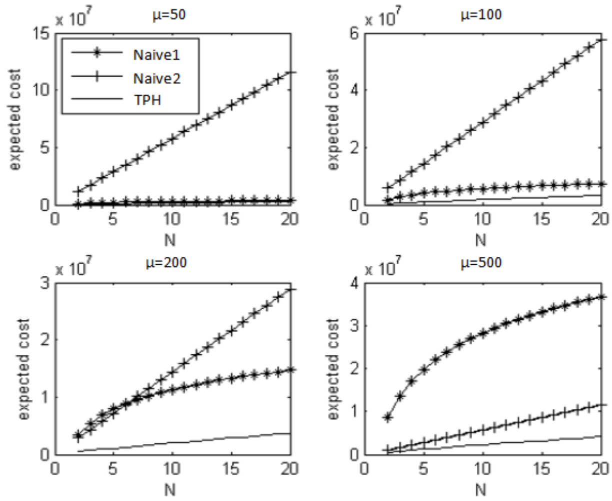


Figure 4: Comparison of our theoretical upper bound on the cost of probe step to Naive1($N1$) and Naive2($N2$) algorithms for $m = 17000$ and different values of $\mu_i = 50, 100, 200, 500$. Y-axis represents the cost of probe step in terms of the expected number or required operations for the given parameters.

Figure 4 compares our theoretical upper bound on the expected cost of the probe step with the same cost for Algorithms Naive-1 and Naive-2. We fix $m = 17000$, number of movies in the Netflix dataset. We use four different values of μ_i and change N from 2 to 20. We use Equation 4 to calculate an upper bound on the expected number of operations for the two-phase algorithm. We use the cost of $log(N)m\mu_i$ for Algorithm Naive-1 and Nm^2/μ_i for Naive-2 (which are the dominant terms of their cost expressions). The average number of ratings in the Netflix dataset for a user is around 200. Figure 4 shows for even very small values of μ_i, the two-phase algorithm works slightly better than Naive-1 while Naive-2 has the worst performance. The performance of Naive-1 gets worse for larger μ_i and the two-phase algorithm performs much better than Naive-1. As discussed earlier, for larger values of μ_i, $Naive2$ has a relatively smaller expected cost (see Figure 4). The majority of the users in the Netflix dataset ($\approx 430,000$), have less than 500 ratings each and more than half ($\approx 270,000$) have between 50 and 500 ratings. In our experimental results, we compare all of the algorithms Naive-1, Naive-2, Hybrid and, the two-phase algorithm.

5.3 Estimating *prob* values

We described our two-phase process using an extension LP of the data structure L by adding the so-called *prob* values. Corresponding to every similarity value s, there is a *prob* value p which is an estimate of the probability of any other similarity value in LP being greater than s. In practice, since a descending ordering of entries in columns of L by similarity is equivalent to an ascending ordering of entries by *prob* values, there is no reason to store the *prob* values, and we can just use the data structure L. Alternatively, the threshold θ on *prob* values could be converted to a similarity threshold.

We estimate *prob* values through fitting a probability distribution to all existing similarity values in L. We consider three different distributions – uniform, Gaussian and Gamma. In the case of Gamma distribution since input values must be positive, we add a constant integer to all similarity values in order to make sure they are all positive.[10] However, in the final weighted sum, we use the original values. Since these are all parametric distributions, we can use similarity values and maximum likelihood estimation (MLE) to find optimal parameters. In the case of uniform distribution, parameter estimation is trivial. We only need to keep minimum and maximum similarity values and no MLE is required. Since the ratings data keeps changing, update is an important issue. Updating the parameters for uniform distribution with respect to the changes in data is straight forward. We just need to keep the minimum and maximum similarity values refreshed. In the case of the other two distributions, sufficient statistics need to be maintained. For instance in the case of Gaussian distribution we need to keep track of mean and standard deviation of similarity values. In the case of Gamma distribution parameter estimation is a little more challenging but still possible. Updating parameters of this distribution is also trivial through keeping sufficient statistics. We refer the reader to (reference here) for details of parameter estimation through MLE for Gamma distribution. In our experiments, we compare the three distributions against each other and also compare the two-phase algorithm with other algorithms. Once we find the optimal threshold θ, we can solve $\theta = 1 - CDF(s)$ for s in order to find the similarity threshold. Here, CDF refers to the cumulative density function of any of the above distributions. For lack of space, we omit the details of the methods for efficiently updating the parameters for the three distributions and refer the reader to [16] for details.

6. UPDATING SIMILARITY MATRIX

Several different similarity measures have been proposed for item-based collaborative problem [1] . Pearson correlation, Cosine and Jaccard are some of the popular examples. Among all of these, Pearson correlation has been applied widely in practice. It is possible to provide guidelines in order to keep the similarity matrix updated for most o these measures. Here, we show how this is doable for Pearson correlation measure since it has been more popular than others. Equation 9, shows how similarity is calculated using Pearson correlation coefficient between two items v_i and v_j.

$$
\begin{aligned}
s(i,j) &= \frac{\sum\limits_{u \in I_{ij}} (R(u,v_i) - \bar{r}_{v_i})(R(u,v_j) - \bar{r}_{v_j})}{\sqrt{\sum\limits_{u \in I_{ij}} (R(u,v_i) - \bar{r}_{v_i})^2 \sum\limits_{u \in I_{ij}} (R(u,v_j) - \bar{r}_{v_j})^2}} \\
I_{ij} &= v_i \cap v_i
\end{aligned}
\tag{9}
$$

It measures the similarity between two items using only ratings of users who have rated both items (I_{ij}). The main idea for partially updating the similarity matrix is as follows:

1. Define the similarity matrix (L) as a materialized view on the top of other matrices containing sufficient statistics for calculating similarity
2. Given one or a set of new ratings, identify which entries of which matrices need to be updated and perform the updates

Equation 10 provides the sufficient statistics that need to be stored in order to be able keep similarities updated.

$$
\begin{aligned}
A_{ij} &= \sum_{u \in I_{ij}} R(u,v_i) R(u,v_j) \\
B_{ij} &= \sum_{u \in I_{ij}} R(u,v_i), C_i = \bar{r}_{v_i} \\
D_{ij} &= \sum_{u \in I_{ij}} R(u,v_i)^2, E_{ij} = |I_{ij}|
\end{aligned}
\tag{10}
$$

Keeping the similarity matrix updated, requires storing four $m \times m$ matrices (A, B, D, E) and a vector of size m (C), containing averages of values as shown in Equation 10. This is in addition to the similarity matrix itself. Since we are keeping similarities in a data structure L and columns of L need to be sorted by similarity, every column of L could be stored as a priority queue. Given the sufficient statistics that Equation 10 provides, we can rewrite and define a materialized view to store similarities in Equation 11.

$$
s(i,j) = \frac{A_{ij} - C_j B_{ij} - C_i B_{ji} + E_{ij} C_i C_j}{\sqrt{(D_{ij} + E_{ij} {C_i}^2 - 2 C_i B_{ij})(D_{ji} + E_{ij} {C_j}^2 - 2 C_j B_{ji})}}
\tag{11}
$$

Storing the similarity matrix as materialized view has its own challenges in practice. In this work, we just provide guidelines for keeping the matrix updated and our focus is more on top-k algorithms. We will study this problem in detail in our future work.

7. EXPERIMENTS

7.1 Data Set and Experimental Setup

In this section, we report the performance of our algorithm as well as the two naive algorithms with respect to several parameters. We use the Netflix dataset with 100M ratings from approximately 500K users on 17,770 movies. The rating values range from 1 to 5. All of the experiments were done on a Linux machine with 32GB of main memory and 2.93GHZ-8MB Cache CPU. All algorithms were implemented in C.

As discussed earlier, we use Pearson correlation coefficient as our similarity measure. We further normalize every row of the similarity matrix before constructing the data structure L, as follows. We first subtract the mean value of the row. Then we do a simple normalization using $s := (s - min)/(max - min)$, where max and min represent the maximum and minimum similarities in the row. We do this in order to make sure higher similarity values in some rows do not affect the whole distribution. It is important to note that this operation does not change the order of nearest neighbors in the columns of L, for this property is crucial for algorithms other than Naive-1. Finally, we use the original similarity values rather than the normalized ones for predicting unknown scores.

[10] Recall, Pearson correlation coefficient lies in $[-1, 1]$.

7.2 Choosing the Best Distribution

Recall, *prob* values associated with entries in L are not physically materialized but are meant to be inferred by fitting the existing similarities in L to an appropriate distribution. As mentioned in Section 5.3, we use MLE to fit a probability distribution to the similarity matrix. In order to choose the best distribution, we experimented with uniform, Gaussian, and Gamma distributions. To measure their effectiveness, we randomly chose 100 users and measured the average performance of the two phase algorithm, with each of the three different distributions used for estimating the *prob* values.

According to our empirical results, and as Fig. 5 shows, we found Gaussian distribution to be the most suitable according to its performance. Uniform distribution provided competitive results. However, we found Gamma distribution completely uncompetitive. Thus, in the rest of our experiments, we only report our results for the two-phase algorithm, based on estimating *prob* values using Gaussian distribution.

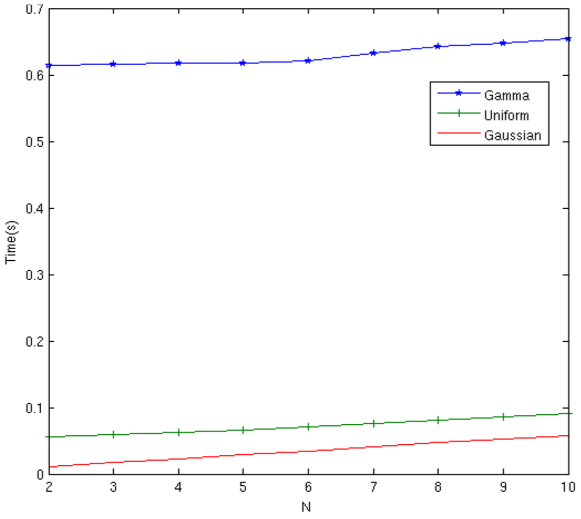


Figure 5: Performance of the Two-phase algorithm using different distributions for estimating *probe* values

7.3 Algorithms Compared

Next, we ran tests to compare the two-phase algorithm with Algorithms Naive-1, Naive-2, and Hybrid. Using the same random set of users as used above for gauging the distributions' effectiveness, we measured the average performance of all algorithms. We preferred to use a random set of users for this purpose since it has the advantage of not being limited to any value (or narrow range of values) of μ_i.

Figure 6 shows the relative performance of Algorithms Naive-1, Hybrid, and Two-Phase, for finding the top-10 recommendations (i.e., $k = 10$). We found Naive-2 much slower than other algorithms on average, making its inclusion in the figure difficult. To put this in perspective, when $N = 10$, while Naive-1 took about 0.08s, Naive-2 took 3.7s, or about 45 times worse. Recall, in Sec. 3, we theoretically derived the "switchover" value of μ_i for the Hybrid algorithm to switch from Naive-1 to Naive-2. We empirically found a different switchover value was more effective for making the performance of Hybrid more competitive. We found that if we use Naive-1 for any user who has rated less than 1500 ratings and Naive-2 for the rest, Hybrid's performance becomes most competitive. We note that 1500 is larger than the theoretical switchover value of μ_i provided in Sec. 3. This is due to the fact that in many cases, the performance of Algorithm Naive-1 is better than its worst case running time. This is expected because the tentative list of top N most similar items maintained by the algorithm does not need to get updated every time during the execution. We empirically found that Naive-2 starts getting more efficient than Naive-1 for values of μ_i greater than 1500. As can be seen in Fig. 6, the two phase algorithm outperforms all algorithms on average. Comparing this to Fig. 4 confirms our theoretical results that our algorithm have a better expected cost compared to naive algorithms.

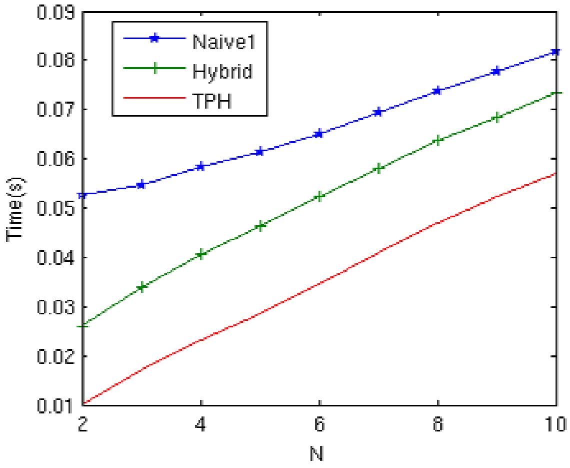


Figure 6: Average performance of Naive-1, Hybrid and the Two-phase algorithm (TPH) on a randomly selected set of users.

7.4 Probe versus Explore

However, the gap between the algorithms does not too much increase with N. We justify this by the same observation we made about hybrid algorithm earlier. An efficient implementation of the Naive-1 algorithm does not need to compare every item on the list of rated items to all of the current top N items during execution. Therefore, Naive-1 performs better than expected in practice. It is also worth mentioning that the execution cost shown in both of the figures corresponds only to the probe step. The cost of explore step is the same for all algorithms. We found the explore step typically takes less than a millisecond in our experiments for $N = 10$. This confirms the validity of our argument about probe being the dominant component of cost.

7.5 Scalability

A brief discussion of what scalability means in this context is in order. First of all, since we assume similarities are precomputed, the number of users will make no difference to the running time of any of the algorithms. We found that all the algorithms discussed scale linearly w.r.t. number of items in the system, even though as already observed, the two-phase algorithm outperforms other algorithms. Thus, we decided to measure scalability of algorithms w.r.t. the average number of items rated by a user, i.e., the average size of the profile of an active user. Given the different approaches adopted by the various algorithms, we expect the average profile size to reflect their relative strengths and weaknesses as well as the tradeoff between Naive-1 and Naive-2.

Thus, we chose 5 different ranges of values for μ_i. We randomly chose 5 users from the Netflix dataset who have rated a number of items in each of the ranges. Fig. 7, shows average performance of the algorithms for each value.

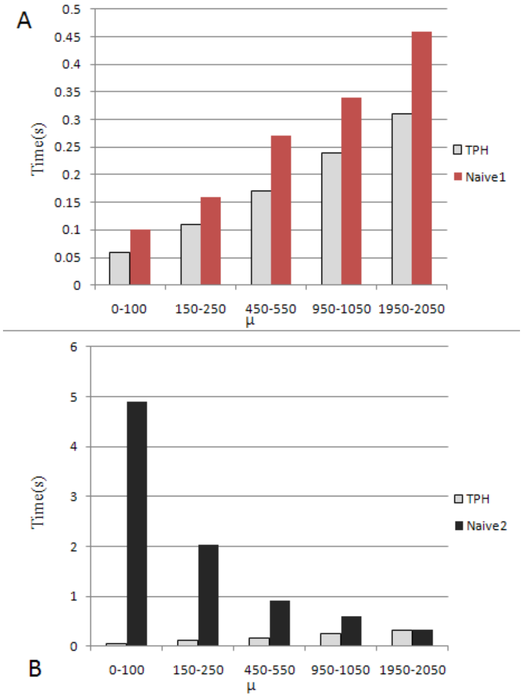


Figure 7: Performance of different algorithms with respect to different values of μ.

We can see that for small values of μ, Algorithm Naive-2 is very inefficient. The two-phase algorithm outperforms Naive-1 algorithm in all cases. The gap between Naive-1 and the two-phase algorithm increases with μ_i. Naive-2 becomes more efficient than Naive-1 as expected for large values of μ_i. However, even for a considerably large value of μ_i such as 2000, which is significantly in favor of Naive-2, and is extremely rare in the Netflix data set, we found that the two-phase algorithm performs as good as Naive-2. This confirms the scalability and reliability of the presented two-phase approach. This should be contrasted with our technical results in Section 4 showing that for any values of the parameters, the classic top-k algorithms can have an unpredictable performance, in that on some instances they access as many entries as the naive algorithms.

7.6 Cost-based Optimization

Finally, in order to evaluate the significance of our cost-based optimization, we choose 4 different ranges of values for μ_i. Again, in each range we found 5 users and found their top-k recommendations. We used $N = 10$ and ran the two-phase algorithm with changing similarity threshold values as well as the optimal value obtained from cost-based optimization algorithm.

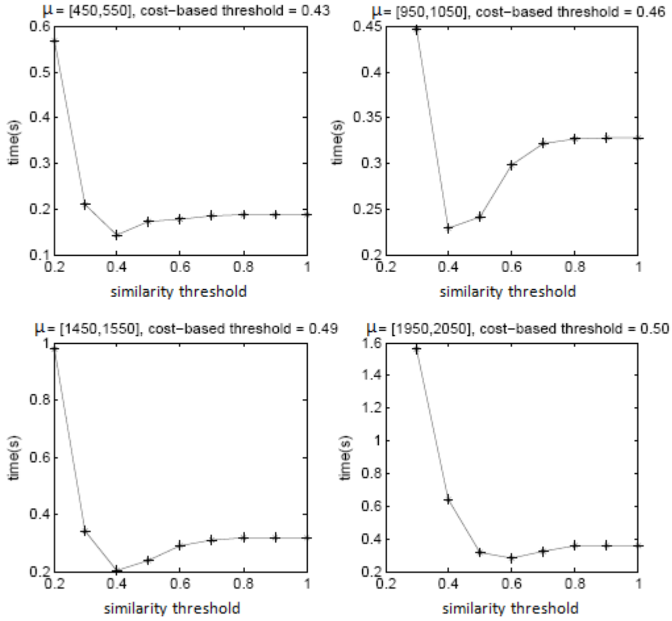


Figure 8: Performance of TPH with respect to different values of similarity threshold, for different values of μ.

Fig. 8 shows the running time of algorithm using different similarity threshold values. It can be seen from this figure that as argued and shown in Section 5, the running time also follows the same shape as the cost function defined in that section. We can see in all cases, the curve has only one minimum. This is due to the tradeoff that exists between the two phases of the algorithm. We show the optimal threshold value resulting from our optimization approach as well. Although the number of ratings might be different for each of the users in the range, we obtain the threshold using median as μ_i. In some cases, obviously, choosing the right value of the threshold is of more importance. Our selected threshold values are reported in the title of each sub-figure and it can be verified that in all cases the value obtained from our method is quite close to the optimal threshold value. This can be seen best when $\mu_i \in [950, 1050]$ where choosing the right threshold is of more importance due to the shape of the curve. The optimal threshold value in this case is between 0.4 and 0.5. Our algorithm finds 0.46 as the best threshold value which actually results in a running time faster than both of the points examined here. Using 0.46 as similarity threshold in this case, the two-phase algorithm finds N nearest neighbors of all items in 0.21(s).

8. CONCLUSIONS

While tremendous strides have been made in recommender systems in developing and tuning algorithms with high accuracy of prediction, the issue of scalability, particularly for finding top-k recommendations for an active user, as opposed to predicting the scores of items, has received relatively less attention. This is the primary focus of this paper, where we concentrate on item-based collaborative filtering, a popular approach in recommender systems that boasts high accuracy of prediction. We show that direct adaptations to classic top-k algorithms such as the TA/NRA family leads to algorithms which either require unrealistic preprocessing and storage or end up accessing as many entries from similarity lists as certain naive algorithms on some instances, *regardless of the problem parameters*. We develop a novel approach based on abstracting the work required for finding top-k recommendations as two key operations – probe and explore, the former by far being the expensive one. Our approach is to use a similarity (or probabilistic) threshold that cuts down the number of entries accessed by the algorithm where the threshold is chosen in order to optimize the expected cost of the algorithm. We prove the cost function has a unique minimum which greatly facilitates the use of numerical approaches in determining the optimal threshold. We demonstrate using extensive experiments on the Netflix data set that the algorithm we propose for top-k recommendations based on item-based collaborative filtering is highly scalable and analyze its relative performance compared to other algorithms for various choices of parameters.

Several problems remain open. The data in a recommender system is subject to frequent change. It is important to update the similarities as well as the *prob* values. In [16], we describe some key ideas for maintaining the distributions over the similarity matrix refreshed, by maintaining sufficient statistics. It would be interesting to also develop an efficient algorithm for incrementally updating the similarity values. Finally, developing top-k algorithms for other recommender approaches such as model-based methods is an important direction for future work.

9. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Towards the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. IEEE TKDE 17(6): 734-749, 2005.

[2] J.S. Breese et al. ŞEmpirical Analysis of Predictive Algorithms for Collaborative Filtering. UAI 1998.

[3] K. Chang et al. Minimal probing: supporting expensive predicates for top-k queries. SIGMOD 2002.

[4] A. Das et al. Google news personalization: Scalable online collaborative filtering. WWW 2007, pages: 271-280.

[5] Mukund Deshpande et al. Item-Based Top-N Recommendation Algorithms. ACM Transactions on Information Systems. Vol. 22, No. 1, 2004, pages: 143-177.

[6] R. Fagin et al. Optimal Aggregation Algorithms for Middleware. Journal of Computer System Sciences. Vol. 66, 2003, pages: 614-656.

[7] Joseph A. Konstan: Introduction to recommender systems. SIGMOD 2008.

[8] Y. Koren. Tutorial on Recent Progress in Collaborative Filtering, RecSys 2008.

[9] D. Knuth. The Art of Computer Programming. Vol. 3, Fourth Edition, Addison-Wesley 2005.

[10] Y. Luo et al. SPARK: Top-k Keyword Query in Relational Databases, SIGMOD 2007.

[11] Movielens. http://movielens.umn.edu.

[12] Netflix. http://www.netflix.com.

[13] P. Resnick et al. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. ACM Conf. Computer Supported Cooperative Work, 1994.

[14] Badrul Sarvar et al. Item-Based Collaborative Filtering Recommendation Algorithms. WWW 2001, pages: 285-295.

[15] Panagiotis Symeonidis et al. Collaborative Recommender Systems: Combinint Effectiveness and Efficiency. Expert Systems with Applications. Vol. 34, 2008. pages: 2995-3013.

[16] M. Tajer and L.V.S. Lakshmanan. TopRecs: Top-k Algorithms for Item-based Collaborative Filtering. Tech. Report. Dept. of Computer Science, UBC September 2010. http://www.cs.ubc.ca/ laks/topK-techReport.pdf.

[17] M. Theobald et al. Top-k Query Evaluation with Probabilistic Guarantees. VLDB 2004, pages: 648-659.

[18] D. Xin et al. Progressive and Selective Merge: Computing Top-K with Ad-hoc Ranking Functions. SIGMOD 2007.

[19] K. Yu et al. Instance Selection Techniques for Memory-Based Collaborative Filtering, Proc. Second SIAM IntŠl Conf. Data Mining (SDM), 2002.

[20] K. Yu et al. Probabilistic Memory-Based Collaborative Filtering. IEEE TKDE, vol. 16, no. 1, pp. 56-69, Jan. 2004.

[21] T. Wu et al. ARCube: Supporting Ranking Aggregate Quaries in Partially Materialized Data Cubes. SIGMOD 2008.