Efficient Algorithms for Recommending Top-k Items and Packages

Mohammad Khabbaz, Min Xie, Laks V.S. Lakshmanan
Department of Computer Science, University of British Columbia
{mkhabbaz,minxie,laks}@cs.ubc.ca

ABSTRACT

Recommender systems use methods such as collaborative filtering in order to make personalized recommendations to users. Collaborative filtering has become a prominent approach for making predictions and personalized ranking. It forms the core business of companies such as Amazon and Netflix. Many sophisticated algorithms have been proposed and much effort has been devoted to improving the accuracy of predictions. Most of this research has been concerned with what we regard as *first generation recommender systems*. Ever since the database community got interested in recommender systems, people have begun researching questions related to flexibility and functionality. In this paper, we propose an integrated framework to review and explain some of the recent work conducted by our group to address some of these questions. We provide efficient methods for updating recommendation models and computing top-k recommendations. In addition to recommending individual items, we propose methods to recommend packages subject to user defined constraints. We refer to this integrated framework as TopRecs+. Our goal is to design an efficient, scalable and flexible recommender system for the next generation suite of applications.

1. INTRODUCTION

Recommender systems use methods such as collaborative filtering in order to make personalized recommendations to users. Collaborative filtering has become a prominent approach for making predictions and personalized ranking. It forms the core business of companies such as Amazon and Netflix. Many sophisticated algorithms have been proposed and much effort has been devoted to improving the accuracy of predictions[1]. Most of this research has been concerned with what we regard as *first generation recommender systems*.

Ever since the database community got interested in recommender systems, people have begun investigating questions related to efficiency, scalability, flexibility and functionality [6, 17]. Following this line of work, our group in UBC has initiated several projects to push the envelope on recommender systems by considering flexible recommender systems which can efficiently compute top-k items within their framework [16] and using recommender systems to design packages subject to user specified constraints [9, 10].

Many of the recommendation algorithms, while highly accurate, have scalability issues. The number of items managed by modern information systems is growing rapidly. Therefore, scalability is one of the serious issues for future generation recommender systems. Recommendation methods try to capture personalized patterns in user feedback data by making assumptions and keeping dense summaries of data. User feedback is typically represented in the form of a sparse matrix that stores existing ratings of users on items. Any item recommendation process has two steps: (1) an off-line training phase that captures personalized profiles (either as a model or as a similarity matrix); (2) an online recommendation generation process that uses the latest up-to-date model or similarity matrix to return top-k recommendations for a user. Any approach for improving scalability of item recommendation needs to pay attention to both profile building and recommendation generation (Section 2). In addition to efficiency and scalability, an important limitation of classical recommender systems is that they only provide recommendations consisting of single items. It has been recognized that several applications call for package recommendations consisting of sets, lists or other collections. E.g., in trip planning [9, 10], a user is interested in suggestions for a set of places (POIs) to visit. If the recommender system only provides a ranked list of POIs, the user has to manually figure out the most suitable set of POIs, which is often non-trivial as there may be a cost to visiting each POI (time, price, etc.), and the user may want the overall cost of all POIs to be less than a cost budget. Furthermore, some additional constraints such as "no more than 3 museums in a package", "not more than two parks", "the total distance covered in visiting all POIs in a package should be ≤ 10 km." might render the manual configuration process even more tedious and time consuming. Another application which needs package recommendation is music list generation [15], where the system needs to recommend users with lists of songs called playlists, and users may have a constraint on the overall time for listening to these songs, and possibly constraints on the diversity of songs in the list. So in these applications, there is a natural need for identifying top-k packages for recommending the users with high quality packages which satisfy all the constraints.

1.1 TopRecs+

We believe that the next generation recommender systems should be efficient, scalable, and flexible enough to be tailed to different applications and users' customization requests such as the ability to compose packages and other collections, and enforce user-specified constraints. In Figure 1, we show the architecture of our envisioned next generation recommender system that we call TopRecs$^+$.

In TopRecs$^+$, the recommendation engine can choose to recom-

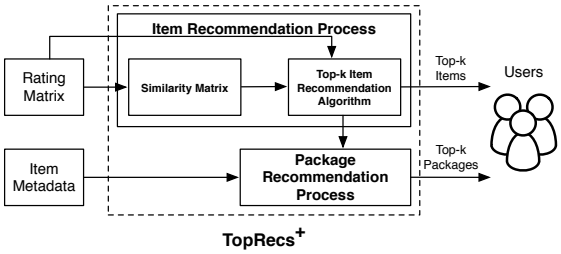

Figure 1: Next Generation Recommender System

mend either top items or top packages depending on the application and user requests. The item recommendation engine can leverage the user item rating matrix to efficiently maintain the item-item similarity matrix, then based on this similarity matrix, an efficient and scalable top-k item recommendation algorithm can provide users with the set of k most interesting items [16]. On the other hand, based on the items generated by the item recommendation engine, combined with metadata information (such as price, type and etc.) associated with each item, the top-k package recommendation engine can return top-k packages that users will be interested in [9, 10].

2. TOP-K ITEM RECOMMENDATION

Predicting personalized scores of individual items for users is the core task of most recommendation algorithms. We follow item-based collaborative filtering (CF) [4], which is used widely in research and in practice [13, 5]. In CF, input data is typically represented as a sparse $n \times m$ matrix(R) of user ratings on items. An entry r_{ij} shows the existing rating of user u_i on item v_j. The main task is to predict the unknown ratings using the existing ones. Item-based CF computes and maintains an item-item similarity matrix using the existing ratings in R. Pearson correlation coefficient [11], is one of the popular choices for calculating item similarities. In item-based CF, the unknown rating $\widehat{r}_{ij}$, of u_i on v_j, is predicted by taking the weighted average of ratings of u_i on N most similar items to v_j. Equation 1 shows how existing ratings are aggregated to calculate $\widehat{r}_{ij}$ where $N(v_j, u_i)$ denotes the set of N items that are most similar to v_j and are rated by u_i.

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

A unique challenge here in providing a score sorted list of items, is the fact that the individual scores to be aggregated for calculating $\widehat{r}_{ij}$ *come from different lists for different items.* Candidate item can have a different set of N nearest neighbors among those rated by u_i and this makes adaptations of classical top-k algorithms inefficient. Our theoretical results in [16] show that adapting classic TA/NRA [12] algorithms which are known to be instance optimal in the classical setting, leads to unpredictable performance due to the above challenge. Therefore, we identify the problem of discovering $N(v_j, u_i)$ for all candidate items to be the costly step in score prediction. For this purpose we propose a novel algorithm called the *Two Phase Algorithm (TPH)* to overcome this challenge.

2.1 Two Phase Algorithm

A naive approach for finding N nearest neighbors of a candidate item v_j in u_i's profile is to retrieve similarities of v_j to all (μ_i) items rated by u_i. Going over all μ_i similarity values and finding the N highest ones can be done in $O(\mu_i log N)$ for one item. Thus, the total cost is $O(m\mu_i log N)$, for all items which can be costly in practice if μ_i is large.

In order to design a more efficient process, we propose a new global data structure, L in place of the similarity matrix. Every column of L corresponds to one of the items. Items in each column are sorted using their similarities with respect to the item indexing the column. Thus, the j^{th} entry in the i^{th} column of L is a pair (item-id, sim), where item-id is the id of the j^{th} most similar item to the i^{th} item. The second element of the pair is the similarity between these two items.

The main intuition behind the two phase algorithm is the following. Assuming that some $N' < \mu_i$ nearest neighbors of a candidate item v_j are known, finding N nearest neighbors can be done more efficiently. This is regardless of whether N' is greater or smaller than N. Suppose we know $N' < N$ nearest neighbors of v_j, then finding the remaining ones can be done in $O(\mu_i log(N - N'))$. If $N' = N$, no further processing is needed. For $N' > N$, it again takes $O(N' log N)$ to find the N nearest neighbors.

All of the required similarity values for finding N nearest neighbors are in the μ_i columns of L that correspond to rated items. Therefore, we propose our two phase algorithm as follows. In the first phase, we choose a similarity threshold and read only those values from these columns that are above the threshold. This leads to discovering a variable number of nearest neighbors for every candidate item. Depending on the number of neighbors found for each item, in the second phase we find the exact N nearest neighbors. For those items that we have managed to find some neighbors for, the process will be more efficient as described earlier.

Figure 2 illustrates the process using a threshold value $\theta = 0.72$. In the first phase all of the entries above the threshold are read as shown on the left side. In this example $\mu_i = 3$. Notice that for both cases of $N = 1$ or 2, the process can be done more efficiently for three out of four candidate items.

2.2 Optimal Threshold θ

The cost of the two phase algorithm ($C(\theta)$) can be written as the sum of three main components: (1) Cost of the first phase ($C1(\theta)$); (2) Cost of finding N nearest neighbors when $N' < N$ ($C2(\theta)$); (3) Cost of finding N nearest neighbors when $N' > N$ ($C3(\theta)$).

For instance, assuming the example in Figure 2 and $N = 2$, v_5 falls in the second category and v_6 falls in the third category. While for v_4 and v_7, we have already found their 2 nearest neighbors. Using smaller θ results in greater $C1$ and $C3$ and smaller $C2$. This is due to the fact that more similarity values will pass the filter and make it to the second phase. On the other hand, $C2$ increases if we use a larger threshold and the other two components decrease. Therefore, there is a tradeoff between $C1$ and $C3$ on one hand and $C2$ on the other. Optimal threshold value is one that minimizes the total cost of all components put together.

We perform a probabilistic cost-based analysis in [16], in order to find the optimal threshold value. In [16], we fit a Gaussian probability distribution to the similarity values in the similarity matrix. Using the cumulative density function, we calculate the probability that a particular similarity value can result in one of the N nearest neighbors of another item. Our cost function provides an upper bound on the expected cost of both phases together. We find the optimal similarity threshold that minimizes $C(\theta)$. Moreover, we theoretically prove that due to the tradeoff between cost components, $C(\theta)$ always has one and only one minimum. We refer the reader to [16] for more details, where we also empirically evaluate our algorithm. Our empirical results confirm the reliability of our theoretical probabilistic process for finding the optimal threshold value which in turn leads to a consistently efficient performance of the top-k recommender algorithm. Figure 3, shows part of our experimental results that shows the effectiveness of the threshold chosen using our method on the performance of the two phase algorithm. TPH takes as input a similarity threshold and performs

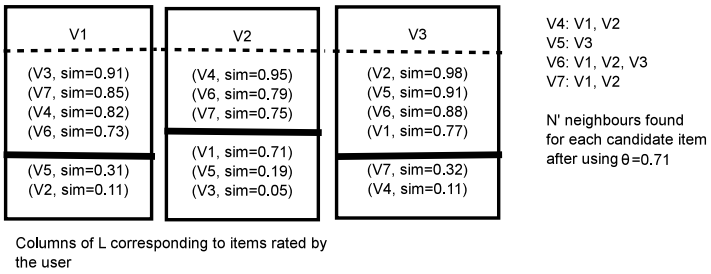

Figure 2: An example of running the two phase probe step using a prob threshold of $\theta = 0.72$ and comparing it to naive algorithms

its two steps according to this threshold as illustrated by Figure 2. In this experiment we vary this threshold from 0.2 to 1 increasing by 0.1 in each step. We also measure the performance using our theoretically found optimal threshold circled in the figure. We find that our theoretically found value for all sizes results in a reliable performance. Particularly, in one case ($\mu \approx 1000$) TPH performed more efficiently using our value compared to the other tested values. The shape of figures obtained by this experiment also highlights the correctness of our results regarding uniqueness of the minimum. We refer the reader to [16] for more details.

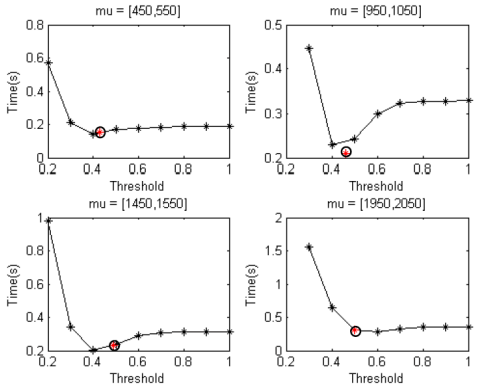

Figure 3: Performance of TPH, using different values of θ.

2.3 Updating the Similarity Matrix

Several measures have been proposed for calculating item similarities the most popular of which is Pearson correlation. It is possible to provide guidelines in order to keep the similarity matrix updated for most of the measures. Here we show the process for Pearson correlation. Equation 2 shows how similarity between two items v_i and v_j is calculated using Pearson correlation coefficient. It measures the similarity between two items using only ratings of users who have rated both items (I_{ij}).

$$s(i,j) = \frac{\sum_{u \in I_{ij}} (R(u,v_i) - \bar{r}_{v_i})(R(u,v_j) - \bar{r}_{v_j})}{\sqrt{\sum_{u \in I_{ij}} (R(u,v_i) - \bar{r}_{v_i})^2 \sum_{u \in I_{ij}} (R(u,v_j) - \bar{r}_{v_j})^2}}, I_{ij} = v_i \cap v_i \quad (2)$$

When a new rating becomes available by u_i, there are μ_i entries in the similarity matrix that need to be updated. Computing all of these values from scratch can be costly. It is more efficient to break down the Equation 2 into a number of components that can be partially updated. Figure 4, shows these partially updatable components. More specifically, four matrices are required which have the same size as the similarity matrix and one vector maintaining average rating of each item. Figure 4, shows how we can rewrite an entry of the similarity matrix using these components.

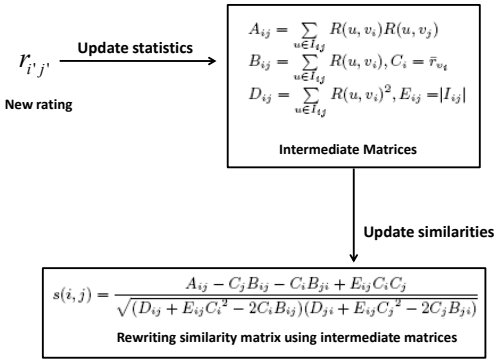

Figure 4: Updating the similarity matrix with $r_{i'j'}$.

As an example it is well-known how to update the average of a list of numbers iteratively. Given that we know the previous average rating C^n of a user. When the new rating becomes available we can update $C^{n+1} = (\mu \times C^n + r_{i'j'})/(\mu + 1)$. Here C^n represents the average rating of the user before the new rating is added to R and C^{n+1} is the updated average. While we know μ is the number of ratings of the user before the addition of $r_{i'j'}$. Performing updates to the other components also follows a similar procedure. Providing efficient strategies for performing these updates in terms of computational cost, quality of results and memory requirements is part of our ongoing work.

3. TOP-K PACKAGE RECOMMENDATION

As mentioned in the introduction, many applications like trip planning and music list generation can benefit from having packages recommended instead of a ranked list of single items.

Let $\mathcal{I}$ be the set of all items, for each item $v \in \mathcal{I}$, we denote the *value* of v for the current active user as $val(v)$ which can be obtained as the predicted utility or rating from the underlying item recommendation algorithms. We denote the *cost* of v as $c(v)$. The cost may correspond to time, price, etc. For a subset of items $P \subseteq \mathcal{I}$, we define the *value* of P as $val(P) = \sum_{v \in P} val(v)$, and the *cost* of P as $c(P) = \sum_{v \in P} c(v)$. Let $\mathbf{P} = \{P \mid P \subseteq \mathcal{I}\}$ be the set of all possible subsets of items, and given a user defined cost budget B, a subset of items $P \subseteq \mathcal{I}$ is *feasible* if $c(P) \leq B$. We can define our top-k package recommendation problem as follows.

DEFINITION 1. *(Top-k Package Recommendation): Given a universe of items $\mathcal{I}$ and an underlying item recommender system for predicting values of items for the current active user, a cost budget B, find top-k packages $P_1, ..., P_k$ such that each P_i is feasible*

($c(P_i) \leq B$) and $val(P) \leq val(P_i)$ for all feasible packages $P \in \mathbf{P} - \{P_1, ..., P_k\}$.

For some applications, the order of items in the recommended package is important, and in addition to the cost associated with each item, we may also need to consider the cost between consecutive items. E.g., for trip planning, we need to consider the time spent on traveling between corresponding POIs. In [10], for each item pair (v_1, v_2), we denote the cost of having v_2 follow v_1 in the recommended package as $c(v_1, v_2)$, and given a set of items V, we define $sc(V)$ as the cost of the minimum walk which covers all items in V.

DEFINITION 2. *(Top-k Sequence Recommendation): Given a universe of items $\mathcal{I}$ and an underlying item recommender system for predicting values of items for the current active user, a cost budget B, find top-k packages $P_1, ..., P_k$ such that each P_i is feasible ($c(P_i) + sc(P_i) \leq B$) and $val(P) \leq val(P_i)$ for all feasible packages $P \in \mathbf{P} - \{P_1, ..., P_k\}$.*

As discussed in [9], when $k = 1$, the top-k package recommendation problem can be viewed as a variation of the 0/1 knapsack problem [7], where we have the restriction that items can be accessed only in the non-increasing order of their value. This is because of the way recommendations are made by the underlying item recommender system. Similarly, the top-1 sequence recommendation problem can be viewed as a variation of the orienteering problem [2]. Furthermore, because solving the top-k package (sequence) recommendation problem optimally is NP-hard [7], we need to consider approximate answers instead of exact ones, i.e., in Definition 1 and Definition 2, for a package P_i in the top-k package set, instead of requiring $val(P) \leq val(P_i)$ for all feasible packages $P \in \mathbf{P} - \{P_1, ..., P_k\}$, we aim for $val(P) \leq \alpha \times val(P_i)$, where α is the approximation factor.

Let c_s be the access cost of retrieving the next highest-value item from the underlying item recommender system and let c_r be the access cost of obtaining the cost (time, price etc) associated with an item or an item pair. It is clear that total access cost of processing n items is $n \times (c_s + c_r)$. Notice that c_s and c_r can be large compared to the cost of in-memory operations: for both accesses information may need to be transmitted through the Internet, and for the sorted access, $val(v)$ may need to be computed. So well-known algorithms for knapsack/orienteering which need to access *all* items [7] may not be realistic, and instead we should minimize the total number of items accessed by the algorithm and yet ensure that high quality top-k packages are obtained.

In [9] we also show that without background knowledge about the cost distribution of items, in the worst case, we must access all items to find top-k packages. To facilitate the pruning of item accesses, we thus assume some simple background information $\mathcal{BG}$ about item costs. We assume $\mathcal{BG}$ is the minimum item cost for illustrative purposes, however, more sophisticated stats like histogram can be easily incorporated.

In [9] and [10], we have shown that instance optimal algorithms which correctly return approximate top-k packages and minimize access cost are possible. However, these algorithms may need to leverage on either pseudo-polynomial algorithm or exponential time algorithm, which may lead to high computational cost. To remedy this, we proposed more efficient algorithms which utilize simple greedy heuristics to form a high quality package from the accessed items. Similar to the instance optimal algorithm, this greedy algorithm will always generate a correct approximation to the optimal solution, however, it is not instance optimal among all approximation algorithms with the same constraints and background information.

For many applications of package recommendation, the users might have some additional constraints, e.g., for trip planning, the user may require the returned package to contain no more than 3 museums. To capture these constraints in our algorithms, we can define a Boolean *compatibility function* $\mathcal{C}$ over the packages under consideration. Given a package P, $\mathcal{C}(P) = true$ iff all constraints on items in P are satisfied. We can add a call to $\mathcal{C}$ in the proposed algorithms after each candidate package has been found. If the package fails the compatibility check, we just discard it and search for the next candidate package.

It is worth noting that many constraints studied in the previous work such as [3] and [8] are restricted classes of boolean compatibility constraints. However, depending on the application needs, for scenarios where only one specific type of constraint is considered, e.g., having one item from each of 3 predefined categories, more efficient algorithms like Rank Join [14] can be leveraged.

4. SUMMARY AND OPEN PROBLEMS

In this paper we presented TopRecs+, a functional and practical recommender system with package recommendation capabilities. While we have investigated some initial efforts in realizing our envisioned next generation recommender system [16, 9, 10], much remains to be done for realizing our vision. Our future work on TopRecs+ includes implementing efficient strategies for updating the similarity matrix, approximating top-k results with probabilistic guarantees, adding support for model-based methods in Toprecs+ and taking into account quality measures such as diversity for package recommendation.

5. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward 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] Chandra Chekuri and Martin Pál. A recursive greedy algorithm for walks in directed graphs. In *FOCS*, 2005.

[3] A. Angel et al. Ranking objects based on relationships and fixed associations. In *EDBT*, 2009.

[4] B. Sarvar et al. Item-based collaborative filtering recommendation algorithms. In *WWW*, 2001.

[5] C. Yu et al. It takes variety to make a world: Diversification in recommender systems. In *EDBT*, 2009.

[6] G. Koutrika et al. Flexrecs: expressing and combining flexible recommendations. In *SIGMOD Conference*, pages 745–758, 2009.

[7] H. Kellerer et al. *Knapsack Problems*. Springer, 2004.

[8] M. De Choudhury et al. Automatic construction of travel itineraries using social breadcrumbs. In *Hypertext*, 2010.

[9] M. Xie et al. Breaking out of the box of recommendations: From items to packages. In *ACM RecSys*, 2010.

[10] M. Xie et al. Comprec-trip: A composite recommendation system for travel planning. In *ICDE*, 2011.

[11] P. Resnick et al. Grouplens: An open architecture for collaborative filtering of netnews. In *CSCW*, 1994.

[12] R. Fagin et al. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[13] S. Amer-Yahia et al. Group recommendation: Semantics and efficiency. In *PVLDB*, 2009.

[14] J. Finger and N. Polyzotis. Robust and efficient algorithms for rank join evaluation. In *SIGMOD*, 2009.

[15] D. Hansen and J. Golbeck. Mixing it up recommending collections of items. In *CHI*, 2009.

[16] M. Khabbaz and L. Lakshmanan. Toprecs: Top-k algorithms for item-based collaborative filtering. In *EDBT*, 2011.

[17] A. G. Parameswaran and H. Garcia-Molina. Recommendations with prerequisites. In *RecSys*, pages 353–356, 2009.