Recommendations For Streaming Data

Karthik Subbian
University of Minnesota
Minneapolis, MN
karthik@cs.umn.edu

Charu Aggarwal
IBM Watson Research Center
Yorktown Heights, NY
charu@us.ibm.com

Kshiteesh Hegde
University of Minnesota
Minneapolis, MN
hegde@cs.umn.edu

ABSTRACT

Recommender systems have become increasingly popular in recent years because of the broader popularity of many web-enabled electronic commerce applications. However, most recommender systems today are designed in the context of an offline setting. The online setting is, however, much more challenging because the existing methods do not work very effectively for very large-scale systems. In many applications, it is desirable to provide real-time recommendations in large-scale scenarios. The main problem in applying streaming algorithms for recommendations is that the in-core storage space for memory-resident operations is quite limited. In this paper, we present a probabilistic neighborhood-based algorithm for performing recommendations in real-time. We present experimental results, which show the effectiveness of our approach in comparison to state-of-the-art methods.

1. INTRODUCTION

The increasing importance of web-enabled e-commerce applications has led to a greater popularity of recommender systems. In collaborative filtering applications, users provide ratings for various products, and the collective intelligence stored in these ratings is used in order to make recommendations. Many such systems have seen an increasing volume of transactions and ratings, which has led to numerous *computational and scalability challenges*. This increased volume of ratings has been driven by the larger number of users at these sites, and numerous ways in which implicit feedback is received from user activities.

In many social applications, such as *Facebook*, items are highly transient, and new items can appear with time. As the underlying user patterns change, the recommendations can also vary significantly with time. The increasing size of such data necessitates the use of real-time streaming algorithms. The two most commonly used classes of algorithms, namely latent factor models and neighborhood-based methods, are computationally challenging. In particular, the main challenges in scaling up recommender systems to the real-time scenario with the aforementioned algorithms are as follows:

- Neighborhood-based recommender systems require an offline phase, which is generally computationally intensive. It is often difficult to perform the underlying computations in real time. Furthermore, with dynamic ratings, it becomes challenging to scale up such systems, particularly when the amount of available memory is limited. It is crucial to be able to perform all the computations in core because disk access is often not practical in such settings.
- Latent factor models require the factorization of a large matrix of entries. In cases where the matrix changes rapidly over time, it becomes difficult to perform the factorization in limited space.
- The temporal nature of recommender systems is such that all ratings for a particular user are not received simultaneously. This makes the problem more difficult because the sets of both the specified (observed) and unspecified (unobserved) entries change dynamically with time. Clearly, the streaming model needs to keep up with these changes in real time.

In this paper, we present a probabilistic neighborhood-based algorithm for performing recommendations in real-time. Our approach uses a min-hash based scheme to construct the underlying recommendation system. To handle the large volume of streaming data, we propose a probabilistic data structure for approximate and efficient model maintenance. We also establish theoretical bounds on the approximation error of our online model and show that the error reduces exponentially with available memory.

1.1 Related Work

Recommender systems became increasingly popular in the mid-nineties, as new systems such as GroupLens [11] were proposed for recommendation. The user-based collaborative filtering models were one of the earliest models [11]. User-based methods utilize the ratings of *similar* users on the *same* item in order to make predictions. While such methods were initially quite popular, they are not easily scalable and sometimes inaccurate. Subsequently, item-based methods [3, 4] were proposed, which compute predicted ratings as a function of the ratings of the *same* user on *similar* items. More recently personalization of recommendation with collaborative filtering using *offline* min-hash clustering was proposed [21]. There are recent attempts to characterize user, item, and topic relationships for recommendation

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/2983323.2983663

using stochastic process models in a streaming setting [8]. In another related work a similarity search approximation for neighborhood-based model is proposed [17]. A detailed discussion of neighborhood-based methods for recommender systems may be found in [5, 2]. Latent factor models have become popular in the context recommender systems [6, 10] and some of them have been used in combination with neighborhood methods [7]. Matrix factorization is another popular approach for learning latent factors and online updates for factorization is another trend in this direction [15, 16]. A detailed discussion of recent advances in collaborative filtering, with a specific emphasis on latent factor models, may be found in [9, 2]. A significant amount of work has been devoted to mining algorithms for streaming data [1]. In this paper we directly approach the problem of recommendation in a streaming setting using a probabilistic neighborhood model with a min-hash technique.

2. RECOMMENDATION FRAMEWORK

In this section, we will introduce the framework for the streaming recommendation problem. A major assumption in streaming recommendation applications is that all the ratings of a particular user or all the ratings of a particular item are not received at the same time. This is different from multidimensional streaming applications in which all dimensions of the same record are always received simultaneously. In recommendation applications, a user may specify a rating for a particular item at any given time. Furthermore, a new user or a new item may enter the system at any given time. In general, since ratings are never deleted, the number of users and items may only increase over time, but they may never decrease. Therefore, we make the assumption that the number of users at time t is given by $m(t)$ and items at time t as $n(t)$. Clearly, $m(t)$ and $n(t)$ can non-decreasing with time t. Therefore, at any given time t, the size of the ratings matrix is $m(t) \times n(t)$.

The ratings are received in the format $\langle UserId, ItemId, Rating \rangle$, as users enter ratings over time. While it is reasonable to assume that users will specify a rating only once for an item, it is possible to handle duplicate ratings by retaining only the latest rating entered by a user for a particular item. Let us assume that ratings have binary values, which are drawn from $\{-1, +1\}$. However, non-binary ratings can also be handled in a relatively straightforward generalization of our technique. A value of $+1$ indicates a liking of an item, whereas a value of -1 indicates a dislike.

In the online setting, it may be desirable to query the system at any given time for recommendations. Typical examples of queries are as follows: (a) determine the top-k recommended items given a user, and (b) determine the top-k relevant users given an item. The first form of recommendation is common more in web-centric applications, while the second form is more useful for targeted marketing scenarios. Despite their application centric differences, the two formulations are similar from a conceptual point of view and minor variations of the same approach can be used to solve both of them.

In this paper, we study a neighborhood-based approach to perform the recommendations. The main problem of the neighborhood-based approach is that it requires an offline phase, in which the similarity between every pair of item is computed. This is particularly difficult in the stream setting because the update of a single rating may require the re-computation of the distance between the item and all other items. Furthermore, one cannot assume that the entire ratings matrix is held in main memory. In such cases, the approach may require several passes over the disk just to compute the pairwise item similarity. While it is reasonable to assume that the ratings matrix will be stored on disk, it is important to be able to maintain memory-resident summaries so that *most* of the computations can be performed in main memory. As we will see later, we will design an approach which requires mostly memory-resident operations and limited access to disk.

2.1 Offline Neighborhood-based Model

Before discussing the streaming model in detail, we will first introduce an offline neighborhood-based model. We will focus on item-based models because they are generally more accurate and also particularly well suited to the streaming scenario.

Consider two items i and j, for which only a subset of the users have specified ratings at time t. As discussed earlier, each rating is drawn from $\{-1, +1\}$. Let the set of users who have specified a positive rating of $+1$ for item i be denoted by $P(i, t)$, and let the set of users who have specified negative rating of -1 of item i till time t be denoted by $N(i, t)$. In order to compute the neighborhood distance between the two items, we will compute the distance separately on the basis of their positive ratings and on the basis of their negative ratings. The similarity $S^+(i, j, t)$ between items i and j at time t on the basis of their positively specified ratings is as follows:

$$S^+(i, j, t) = \frac{P(i, t) \cap P(j, t)}{P(i, t) \cup P(j, t)} \tag{1}$$

In other words, the similarity on the basis of only the positive rating is equal to the Jaccard coefficient between the two sets. A corresponding notion can be defined on the basis of the negative ratings $N(i, t)$ and $N(j, t)$ as $S^-(i, j, t)$.

The overall similarity between the two items is then obtained by combining the positive similarity results $S^+(i, j, t)$ and the negative similarity results $S^-(i, j, t)$. However, this combination must appropriately weight the number of positive and the number of negative ratings. Therefore, the combined similarity $S(i, j, t)$ is as follows:

$$S(i, j, t) = \frac{\alpha \cdot S^+(i, j, t) + \beta \cdot S^-(i, j, t)}{\alpha + \beta}, \tag{2}$$

where, $\alpha = |P(i, t)| + |P(j, t)|$ and $\beta = |N(i, t)| + |N(j, t)|$. Here α and β are the weights of the positive and the negative ratings, respectively, for an item-pair combination (i, j) at time t. These similarities are used to define the groups of most similar items.

2.2 Predicting Ratings with Similar Items

The peer groups can be used to predict the specific ratings of particular user-item combinations. The first step is to compute the similarity of a given item with all the other items. The weighted average of the ratings of the most similar items to a particular item i is predicted as the user rating for that item. Let $I_i(u)$ be the set of items most similar to item i, for which the user u has specified ratings. Furthermore, let the specified rating of user u for item j be r_{uj} (assuming that a rating has indeed been specified). Then,

the overall neighborhood-based *prediction* $p_{ui}(t)$ of user u for item i at time t is as follows:

$$p_{ui}(t) = \frac{\sum_{j \in I_i(u)} (S(i,j,t) \cdot r_{uj})}{\sum_{j \in I_i(u)} S(i,j,t)} \quad (3)$$

Note that this computation requires the determination of the most similar items to a given item. This can require disk-resident offline computation. Clearly, this is not a feasible option for resolving online queries. In the static offline setting, the most similar items to a given item are pre-computed a priori, which are used to provide query responses. Therefore, it is crucial to set up both disk-resident and memory-resident data structures, which can be leveraged effectively at query time. Although a limited amount of access to the disk-resident ratings matrix continues to be required, most of our approach uses probabilistic data structures, such as the min-hash index, which can be maintained in an online fashion.

3. STREAMING RECOMMENDATIONS

In this section, we will propose a min-hash technique to perform recommendations in a space-constrained setting. The idea in the min-hash approach is to use a probabilistic data structure, which can approximately compute the similarity between the items.

The similarity can be approximately computed by tracking the relevant users for each item in a min-hash index. Let us consider two items i and j. As before, assume that the set of users who have rated item i positively at time t is denoted by $P(i,t)$ and the set of users who have rated item j positively at time t is denoted by $P(j,t)$. The basic idea is to impose a sort order on the users with the help of a hash function. What is the probability that the first user with a positive rating for item i in this sort order is the same as the first user with a positive rating for item j in the sort order? This is equal to the probability that both columns i and j take on the value of $+1$, when at least one of the columns takes on the value of $+1$. This probability is equal to $\frac{P(i,t) \cap P(j,t)}{P(i,t) \cup P(j,t)}$, which is exactly the same as the Jaccard coefficient between $P(i,t)$ and $P(j,t)$. This is, therefore, equal to the similarity $S^+(i,j,t)$ between items i and j. This basic principle can be used to estimate the Jaccard coefficient by using multiple sort orders to repeat the process and determine the fraction of cases in which the first positive rating for both items is provided by the same user. A similar approach can be used to compute the Jaccard coefficient on the negative ratings. Therefore, the two can be combined in order to compute the similarity between the items. Note that the combination of the similarity on the positive items and the negative items requires the tracking of $|N(i,t)|$ and $|P(i,t)|$, which is performed separately. For each item, we always separately maintain the number of positive and negative ratings of that item. This can easily be maintained in streaming fashion.

How are these random sort orders implemented? The sort orders are implemented with the use of a hash function. In order to implement d mutually independent sort orders, we use d mutually independent hash functions denoted by $f_1(\cdot) \ldots f_d(\cdot)$. The argument of each such hash function is the identifier of a user, and the hash function value is a random number which is uniformly distributed in $(0,1)$. For each item j in the data, the following two pieces of information are maintained:

1. The hash functions $f_1(\cdot) \ldots f_d(\cdot)$ are applied to the user-indices that have rated item j positively. For each hash function $f_s(\cdot)$, the least hash value among these positive users is maintained. Furthermore, the user-index which maps to the least hash value is also maintained. The former is referred to as the *min-hash value*, and the latter is referred to as the *min-hash index*. Because there are d hash functions, a total of d hash function-hash value pairs are maintained for each item. Thus, the sth pair is of the form $(f_s(u), u)$, where u is the index of a user. Overall for $n(t)$ items, a total of $d \cdot n(t)$ pairs are maintained. This data structure maintained over the positively rated items is denoted by $\mathcal{M}^+$.

2. The hash functions $f_1(\cdot) \ldots f_d(\cdot)$ are applied to the user-indices that have rated item j negatively. For each hash function $f_s(\cdot)$, the least hash value among the negative users are maintained. As the previous case, both the hash index and hash value are maintained. This portion of the data structure maintained over the negatively rated items is denoted by $\mathcal{M}^-$.

It is noteworthy that the size of these data structures is much smaller than the size of the ratings matrix, when the value of d is small. Therefore, this data structure can be maintained in main memory.

Furthermore, the data structure can be updated efficiently. Consider an incoming rating r_{ui} that is positive. The first step is to apply the d hash functions to the user u to result in the d hash values $f_1(u) \ldots f_d(u)$. For the ith item, its d different hash value/hash index pairs are retrieved from $\mathcal{M}^+$ because the rating r_{ui} is positive. The sth current hash value (where $s \in \{1 \ldots d\}$) for item i in $\mathcal{M}^+$ is compared to $f_s(u)$. If $f_s(u)$ is smaller than the sth current hash value in $\mathcal{M}^+$, then the corresponding pair in $\mathcal{M}^+$ is replaced with the new pair $(f_s(u), u)$. Otherwise, no change is made to the sth value in $\mathcal{M}^+$ for item i. This process is repeated for each value of $s \in \{1 \ldots d\}$. If the rating r_{ui} is negative, then exactly the same steps are used, except that the changes are made to $\mathcal{M}^-$ rather than $\mathcal{M}^+$.

Note that it is easy to compute the similarity between two item pairs with the use of the min-hash index. For example, the value of $S^+(i,j,t)$ is equal to fraction of the d min-hash indices in $\mathcal{M}^+$, which are the same. In other words, if $i_1 \ldots i_d$ be the top-d min-hash indices of item i in $\mathcal{M}^+$, and $j_1 \ldots j_d$ be the top-d min-hash indices of item j in $\mathcal{M}^+$, then the similarity between items i and j can be estimated as follows:

$$S^+(i,j,t) \approx R^+(i,j,t) = \frac{\sum_{s=1}^{d} \delta(i_s = j_s)}{d} \quad (4)$$

Here $\delta(\cdot)$ is an indicator function, which takes on the value of 1 when i_s and j_s are the same, and 0, otherwise. The value if $S^-(i,j,t)$ can be computed in a similar way, except that the data structure $\mathcal{M}^-$ is used to compute the similarity rather than $\mathcal{M}^+$. In order to combine the similarity measures $S^+(i,j,t)$ and $S^-(i,j,t)$ into a single similarity measure $S(i,j,t)$, the number of positive and negative ratings of the items i and j are required. These can be dynamically tracked in online fashion by incrementing counters tracking the number of positive or negative ratings for each item, as they are received. In addition, the number of specified ratings $N(i,t)$ for each item i is maintained separately.

3.1 Quality of Recommendations

The quality of the recommendation is highly dependent on the quality of similarity computed. Therefore, in the following, we will bound the quality of the similarity, which is computed using the min-hash approach. This is an important result because it ensures a high fidelity for the quality of recommendations.

LEMMA 1 (LOWER TAIL BOUND). *For any $\epsilon \in (0,1)$, Equation 4 provides an approximated value $R^+(i,j,t)$ of the Jaccard coefficient, which lies outside a factor $(1-\epsilon)$ of the true value $S^+(i,j,t)$ with the following probability.*

$$P(R^+(i,j,t) < (1-\epsilon)\cdot S^+(i,j,t)) \leq exp(-d\cdot S^+(i,j,t)\cdot\epsilon^2/2) \quad (5)$$

PROOF. We restate Equation 4 here:

$$S^+(i,j,t) \approx R^+(i,j,t) = \frac{\sum_{r=1}^{d}\delta(i_r = j_r)}{d}$$

$$d\cdot S^+(i,j,t) \approx d\cdot R^+(i,j,t) = \sum_{r=1}^{d}\delta(i_r = j_r)$$

Note that we are summing up d i.i.d. Bernoulli variables in the aforementioned equation. This particular form of the summation can be directly used in conjunction with the Chernoff bound. Each element $\delta(i_r = j_r)$ in the summation is a Bernoulli random variable. Furthermore, this Bernoulli random variable is equal to 1 with probability $S^+(i,j,t)$. In such a case, the lower-tail Chernoff bound applies to the summation. By applying the lower tail Chernoff bound, we obtain the following:

$$\begin{aligned} P(d\cdot R^+(i,j,t) &< (1-\epsilon)\cdot d\cdot S^+(i,j,t)) \\ &\leq \exp(-d\cdot S^+(i,j,t)\cdot\epsilon^2/2) \\ P(R^+(i,j,t) &< (1-\epsilon)\cdot S^+(i,j,t)) \\ &\leq \exp(-d\cdot S^+(i,j,t)\cdot\epsilon^2/2) \end{aligned}$$

This completes the proof. □

LEMMA 2 (UPPER TAIL BOUND). *For any $\epsilon \in (0, 2\cdot e-1)$, the approximated value $R^+(i,j,t)$ of the Jaccard coefficient lies outside a factor $(1+\epsilon)$ of the true value $S^+(i,j,t)$ with the following probability.*

$$P(R^+(i,j,t) > (1+\epsilon)\cdot S^+(i,j,t)) \leq exp(-d\cdot S^+(i,j,t)\cdot\epsilon^2/4)$$

Due to shortage of space, we have omitted the proof for Lemma 2. However, it is straightforward to establish the upper tail bound, with a similar argument to Lemma 1. Thus, the probability of errors in the overall similarity computation reduces exponentially with increasing number of hash functions. This ensures that the approximate approach will yield similar results to the exact approach.

Note that whenever a rating has to be predicted for an user and item pair, it is straight forward to compute the predicted rating using (4), (2), and (3) (in that order). We refer to our proposed approach as STREAMREC in our experiments.

4. EXPERIMENTAL RESULTS

We evaluate our approach in terms of its effectiveness on prediction errors, efficiency using running time and sensitivity to change in model parameters.

4.1 Dataset

We chose two real-world public datasets to cover a different stream lengths, number of items/users, domain diversity, and rating scale.

Book Crossing: A four-week crawl from August to September 2004 from the Book-Crossing community is available at [18]. It contains 278,858 users providing 1,149,780 ratings about 271,379 books. The rating scale in this dataset ranges from 0 to 10.

Dating: This is an online dating website[1], where 17,359,346 ratings of 168,791 profiles are provided by 135,359 website users. All users are anonymized and this data set is publicly available [19]. Online dating website ratings are in the scale of 1 to 10.

These public datasets are stored offline and therefore, we had to simulate the streaming nature. In streaming settings, the user and items pairs mostly arrive in some random order. However, in our data sets the ratings were sorted by users and we account for this by randomizing the order of ratings.

4.2 Baselines

We used several important well-established baselines, which encompass both the neighborhood-based and latent-factor models. The two most popular baselines used in evaluating recommender systems are: Alternating Least Squares (ALS) [12] and Stochastic Gradient Descent (SGD) [10]. Probabilistic Matrix Factorization (PMF) [13] is a Bayesian approach for matrix factorization, in which the model capacity is controlled automatically by integrating over all model parameters and hyper-parameters. SVDPP [7] is another baseline that combines the latent factor models nicely in to the neighborhood based models as an optimization. A gradient descent approach is used to solve the optimization problem. We also evaluated a two-layer undirected graphical model, Restricted Boltzmann Machines (RBM), for collaborative filtering process as another baseline. The parameters for all baselines were tuned using a 5% validation set from the tail end of each data set.

4.3 Evaluation Measure

We have used Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) to quantify the effectiveness of our approach compared to baselines. RMSE penalizes prediction errors quadratically and the measure is affected more significantly by larger errors. For different applications, the importance of these measure may vary. Hence, we evaluate all approaches on both measures.

4.4 Efficiency Analysis

The most important part of a streaming algorithm is its ability to perform model maintenance in online fashion [1]. The basic idea is to be able to deal effectively with high-velocity data sets, by being able to efficiently perform model updates. In this section, we will show the efficiency of maintaining the streaming recommendation model.

We compare the running time of our approach against various baselines in Fig. 1(a) and (d). Our approach is extremely fast in terms of the running time compared to all the baselines. As we progress along the stream, we see newer items or users and more ratings have to be processed. Therefore, as time progresses, all algorithms generally have a larger computational burden. While all baselines take more time to process this, our approach consistently processes 100K records in fraction of a second. This is because of the fast probabilistic approach used for creating the summarized representation in a fixed number of operations. The updates

[1] *http://libimseti.cz/*

used in the probabilistic approach are straightforward, linear, and additive. This makes it ideal for the streaming scenario.

Among the baselines, PMF and ALS were the slowest in terms of the overall efficiency and also in terms of the rate of increase along the progression of stream. This is because PMF uses Markov Chain Monte Carlo (MCMC) approximation to learn the MAP estimate. The use of the MCMC technique is extremely inefficient, particularly in the streaming setting. Both SGD and SVDPP use stochastic gradient updates and are hence relatively fast in term of running time, at least compared to PMF and ALS. Despite their speed, they still have larger running times when they have to process a very large dataset.

Each approach also has an offline querying time, that is the time taken to query the rating for an unseen user-item pair. This time is primarily for offline testing purposes. Our approach took 1.41 milliseconds for bookcrossing and 2.91 milliseconds for the dating datasets. While the fastest baselines SVDPP and SGD required less than 1 millisecond and slowest PMF was about 4 milliseconds on all datasets. In general, these numbers are not distinguishable from a querying latency perspective, because most of the work is really focused in the *streaming* setting of updating the model in an efficient way. All the numbers reported here are using a single CPU. However, testing using our approach is trivially distributable across multiple machines or cores, as long as the M^+ and M^- data structures are available across all machines or in a shared memory.

4.5 Effectiveness Analysis

We used 5% of each dataset for validation and parameter tuning. Of the remaining 95% we use the first 20% of the dataset purely for training our model. We split the remaining data into 20 equal parts (of 3.8%) to check the effectiveness of our approach at those points. The results are shown in Fig. 1(b,c,e,f). To evaluate each of the baselines, we simulate a continuous stream of the training data until a certain point of the stream and test on the next 3.8% of the stream data. Some baselines are of course not incremental in nature. Therefore, to obtain the complete benefit of their approach, they were re-trained from scratch at each evaluation point. In this context, it needs to be pointed out that our approach was evaluated in a far more *restricted setting* compared to the baselines, where such non-streaming "fixes" were not allowed. We chose an arbitrarily small number of hash functions ($d = 20$) for evaluating our approach. As we show later, the effectiveness of our technique increases with increasing the number of hash functions.

As we progress along the stream, all methods clearly benefit from the increasing amount of training data. However, the rate of decrease in RMSE is different for each method as seen in Fig. 1(b,e). When large amounts of training data are available, the decrease is very minimal and the methods are quite stable. This is because the latent factor models do not gain much in terms of additional 3.8% training data. However, our method is quite stable as the similarity values computed do not change until a significant number of new users rate the item.

Another significant benefit of our approach is the much larger change in MAE, while having a stable downward trend in RMSE. Remember that the RMSE values are more affected by a few larger errors. These trends show that the proposed approach does not make increasing number of larger errors, along the stream, while constantly reducing large numbers of smaller errors. While other methods like PMF have stable RMSE, but increasing trend in MAE. These approaches, in a sense, make increasing number of smaller errors while keeping the RMSE under control (which is often their objective).

However, note that our approach works in a restricted streaming setting, whereas the baseline methods are allowed offline retraining from scratch where needed. In spite of this fact, the slightly better performance of a single baseline approach is not very significant.

4.6 Sensitivity Analysis

Our approach is sensitive to the number of hash functions d used in the min-hash approach. Due to lack of space, we have shown the results for only Dating dataset. However, we note that the results were similar for Book-crossing dataset.

As we discussed in Section 3, the reduction in approximation error in computing similarities exponentially falls off with increasing number of hash functions. This is very evident from Fig. 2 (a) for RMSE and (b) for MAE. Thus, a reasonable number of hash function is sufficient in practice to achieve a good performance in terms of RMSE and MAE.

The running time, for our approach, increases linearly with d and this is can be easily seen by looking at the cross-section of plots in Fig. 2(c). It is clear that using smaller number of hash functions is ideal both in terms of effectiveness and efficiency. But this does not mean that a use of $d = 1$ is ideal. In practice, we found that having around 10 to 20 hash functions is reasonable on various datasets.

5. CONCLUSIONS

With the increasing ease in ability to collect implicit and explicit ratings through various manual and automated mechanisms, the importance of streaming recommendations has increased considerably in recent years. In this paper, we propose an efficient algorithm to perform streaming recommendations by using efficient probabilistic data structures. These data structures allow compact representations of the ratings matrix, which can be efficiently leveraged to provide recommendations in online time. Our approach may be considered the first truly online method for performing recommendations in real time settings. Our experimental results validate the effectiveness and efficiency of our approach over existing methods.

6. REFERENCES

[1] C. Aggarwal. Data Streams: Models and Algorithms, *Springer*, 2007.

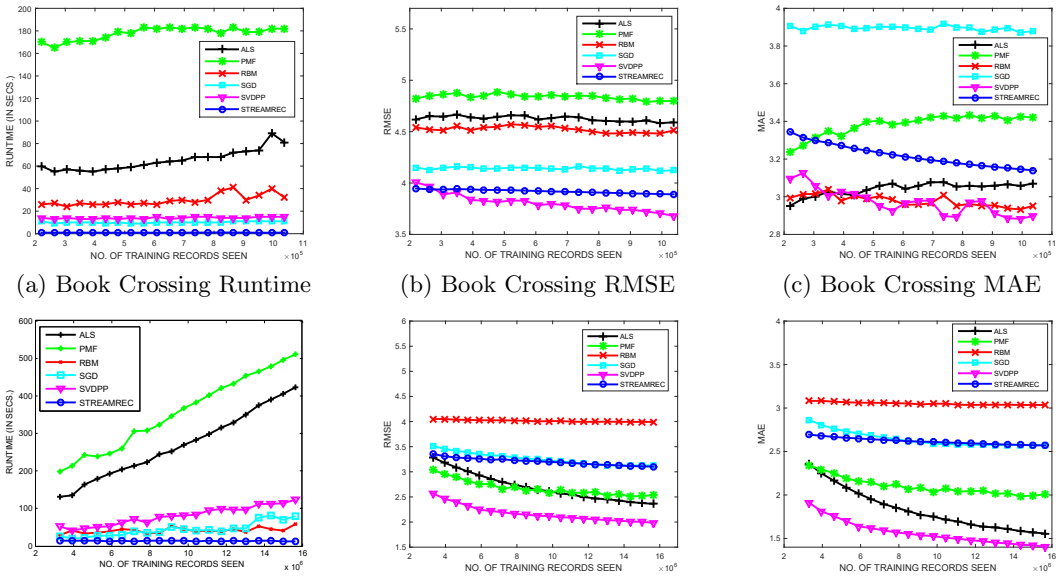
[2] C. Aggarwal. Recommender Systems: The Textbook, *Springer*, 2016.

[3] M. Deshpande, and G. Karypis. Item-based top-n recommendation algorithms. *ACM TOIS*, 22(1), pp. 143–177, 2004.

[4] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. *WWW*, pp. 285–295, 2001.

[5] C. Desrosiers and G. Karypis. A comprehensive survey of neighborhood-based recommendation methods. *Recommender Systems Handbook*, pp. 107–144, 2011.

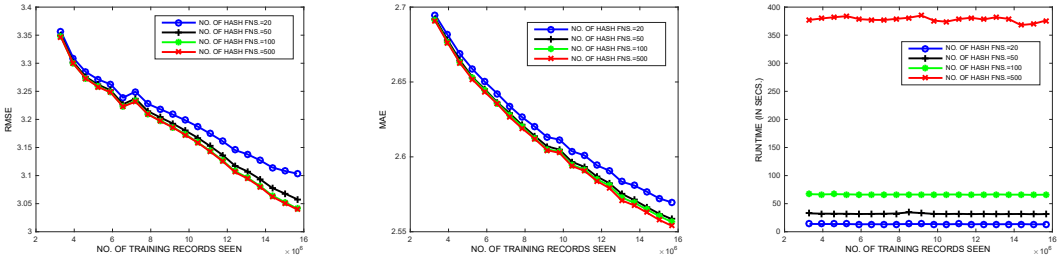
[6] T. Hofmann. Latent semantic models for collaborative filtering. *ACM TOIS*, 22(1), pp. 89–114, 2004.



(a) Book Crossing Runtime (b) Book Crossing RMSE (c) Book Crossing MAE

(d) Dating Runtime (e) Dating RMSE (f) Dating MAE

Figure 1: Efficiency and effectiveness plots for Book Crossing and Dating datasets.



(a) Dating RMSE (b) Dating MAE (c) Dating Runtime

Figure 2: Sensitivity plots for Dating dataset.

[7] Y. Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. *KDD*, pp. 426–434, 2008.

[8] S. Chang, Y. Zhang, J. Tang, D. Yin , Y. Chang, M. H-Johnson, and T. S. Huang. Streaming Recommender Systems, *https://arxiv.org/pdf/1607.06182v1.pdf*, 2016.

[9] Y. Koren and R. Bell. Advances in collaborative filtering. *Recommender Systems Handbook*, Springer, pp. 145–186, 2011.

[10] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8), pp. 30–37, 2009.

[11] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. GroupLens: an open architecture for collaborative filtering of netnews. *CSCW*, pp. 175–186, 1994.

[12] Y. Zhou, D. Wilkinson, R. Schreiber and R. Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. *Algorithmic Aspects in Information and Management*. Shanghai, China. pp. 337-348, 2008.

[13] R. Salakhutdinov and A. Mnih. Bayesian Probabilistic Matrix Factorization using Markov Chain Monte Carlo. *ICML*, 2008.

[14] G. Hinton. A Practical Guide to Training Restricted Boltzmann Machines. University of Toronto, *Tech report UTML TR 2010-003*, 2010.

[15] A. Karatzoglou, A. J. Smola, and M. Weimer. Collaborative Filtering on a Budget, *AISTATS*, pp. 389–396, 2010.

[16] J. Z. Sun, K. R. Varshney, K. Subbian. Dynamic matrix factorization: A state space approach. *ICASSP*, pp. 1897–1900, 2012.

[17] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y Xu. Tencentrec: Real-time stream recommendation in practice, *SIGMOD*, pp. 227–238, 2015.

[18] www2.informatik.uni-freiburg.de/˜cziegler/BX/

[19] www.occamslab.com/petricek/data/

[20] E. Diaz-Aviles, L. Drumond, L. Schmidt-Thieme, and W. Nejdl. Real-time top-n recommendation in social streams. *RecSys*, 2012.

[21] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google News Personalization: Scalable Online Collaborative Filtering. *WWW*, 2007.