Expert Systems With Applications 64 (2016) 84–92

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

An ensemble method for top-N recommendations from the SVD

David Ben-Shimon[*], Lior Rokach, Bracha Shapira

Department of Information Systems Engineering, Ben-Gurion University of the Negev, P.O.B. 653, Beer-Sheva 84105, Israel

ARTICLE INFO

Article history:
Received 20 February 2016
Revised 1 July 2016
Accepted 19 July 2016
Available online 20 July 2016

Keywords:
Recommender systems
Ensemble methods
Matrix factorization
Top-N recommendations
Singular value decomposition

ABSTRACT

Matrix factorization methods such as the singular value decomposition technique have become very popular in the area of recommender systems. Given a rating matrix as input, these techniques output two matrixes with lower dimensional space that represent the user and item features. The relevance of item *i* to user *u* is revealed by the score of the dot product between *u* vector of features and *i* vector of features. High scores indicate greater relevance. In order to deliver the best recommendations for a given user based on these latent features, one must obtain the list of scores of all the items for the given user and sort the resulting list. When the size of the catalogue is large, this phase consumes a large amount of computational time and cannot be done online. Another drawback with this approach is that once such a list is computed for a given user, it remains finite and it is impossible to incorporate within it new activities of the user. Hence, the use of such techniques is limited online.

In this paper we propose an ensemble method for building a forest of trees offline, where each leaf in each tree is holding a unique set of item vectors. Once a user is engaged with the system, its vector is classified to one leaf in each one of the trees in the forest for conducting a dot product with the corresponding items. By using this method we compute online only a small number of dot products for a given user vector allowing us to quickly retrieve dynamic recommendations from the SVD, thereby presenting an alternative to the existing method which computes and caches all of the dot products among the items and users. The method maps the items to the leaves of multiple compact trees offline, each tree is a weak recommendation model, creating a forest of decision trees algorithm in which users that are assigned to these leaves online are likely to produce high dot product scores with the items that are already in the leaves. We demonstrate the effectiveness of the suggested ensemble method by applying it to three public datasets and comparing it to a state-of-the-art algorithm aimed at solving the problem.

1. Introduction

With the growth of e-commerce and its contribution in the economy, recommender systems (RS) have become increasingly important. RSs can play a role in increasing sales, enlarging a company's customer base, converting browsers into purchasers, and more. Once an RS has been provided with a user *u* and an item *i*, it can perform many tasks. Often this begins with predicting the rating that the user will assign to the item, which is one of the most heavily explored issues associated with RSs. For example, in Netflix, the largest online video streaming service, when a user enters the new releases webpage, the system computes the predicted ratings of all newly released movies and presents this information to the user. Another common recommendation task associated with e-commerce applications is to display a list of items – often referred to as the top-N recommendations list – which the system considers to be most relevant for the current user. For example, Amazon presents its users with a list of products referred to as "Recommendations for You in Books" which includes the most relevant books for the user based on Amazon's algorithms. In e-businesses the task of delivering the top-N recommendations is more pertinent than the task of rating items, because e-businesses strive to present interesting items to the user.

When the input data is rating and the task is prediction, an intuitive and popular algorithm applied is an approximation of matrix factorization (MF) based technique (Koren, 2008) called singular value decomposition (SVD). The SVD algorithm decomposes the rating matrix into two low dimensional latent factor matrixes, **p** for users and **q** for items. Once the output **p** and **q** of the SVD algorithm have been learned, the prediction of a rating for a given user *u* and item *i* is computed using Eq. (1).

$$\hat{r}_{ui} = \mu + b_i + b_u + q_i^T \cdot p^u \tag{1}$$

[*] Corresponding author.
E-mail addresses: dudibs@gmail.com (D. Ben-Shimon), liorrk@bgu.ac.il (L. Rokach), bshapira@bgu.ac.il (B. Shapira).

http://dx.doi.org/10.1016/j.eswa.2016.07.028
0957-4174/© 2016 Elsevier Ltd. All rights reserved.

Here μ is the average of the overall ratings, b_i is the bias of item i, and b_u is the bias of user u. $q_i^T \cdot p^u$ is the latent features dot product, capturing the relevance of item i to user u. The matrices **q** and **p** are often learned using a gradient descent approach, minimizing the prediction error. A high value resulted from Eq. (1) indicates high probability that user u will like item i. The success of the algorithm in the area of RSs followed Netflix competition (Bennett & Lanning, 2007) in which the winner was a version of this algorithm. The algorithm was later extended to incorporate various contextual information (Karatzoglou, Amatriain, Baltrunas, & Oliver, 2010).

When the task is top-N recommendations, the SVD algorithm considers the recommendation task as a reduced problem of the prediction task, i.e., the algorithm predicts ratings offline for all items not yet seen by the user and then caches the top-N highest rated items. Such an approach utilizes the predictive power of the SVD, however it has two major drawbacks. First, the catalog of items is often too large to allow exhaustive computation of all of the dot products even in offline mode. For instance, in a system with millions of users and millions of items the process of multiplying user vectors with item vectors yields billions of dot product computations. Second, the user vectors may change as a result of subsequent (even very recent or concurrent) engagement with the system, therefore the need to work offline precludes the ability to work with the most updated user vector which is available only online. Consider a situation in which a list of user specific recommendations has been compiled in advance for the user, after which the user clicked on a few items; in this scenario, the list is incapable of reflecting these clicks and hence is quickly outdated. These two drawbacks, expensive computations and the inability to incorporate the updated user profile in the recommendations, limit the use of the SVD algorithm online when the number of users and the number of items is very large (i.e., in the millions). Therefore, there is a need for a fast online method for computing the recommendations from **p** and **q**.

In this paper, we focus on the ability to retrieve the top-N items from **p** for a given user vector $\mathbf{q}_u$ quickly, without computing all of the dot products or by compiling this list in advance. We suggest an ensemble method for the retrieval of the top-N recommendations from the SVD results that reduces the recommendation time and improves the predictive performance of the original SVD. Generally, ensemble methods do not decrease computational time, however, our method structures the latent features of the items in a forest of small decision trees (each is a weak classifier), thereby causing the computational time to decrease. Each tree is compact and quickly learned, because it's based on a small portion of the latent features. Existing solutions for the given problem are unsatisfactory, often providing less accurate results than the original SVD, and inferior to the suggested method which improves the original SVD and speeds up the computation time.

There are numerous studies (Cremonesi et al., 2010) on how to compute top-N recommendations from MF based method and how to improve the learning process of the features to increase the quality of the recommendation. However, this paper is not suggesting general approach for delivering top-N recommendations from MF based method nor to improve the learning (or factorize) process of the factors, but rather an approach for extracting quickly the list of recommendations out from the already learned (or factorized) features.

The remainder of this paper is organized as follows: Section 2 surveys the current status of algorithms in this area. Section 3 thoroughly describes the proposed method. Section 4 describes the experimental study and the comparative results of the proposed method, and the paper concludes with a discussion in Section 5.

2. Background and related work

Typically, MF based methods for RS such as SVD receive a rating matrix as input and provide the two lower dimensional matrixes **p** and **q** as output. These two matrixes capture the characteristics of the items and users along a preconfigured number of dimensions. Once the matrixes **q** and **p** have been learned, the recommendation engine computes user-specific recommendations by: a) multiplying item factors against user factors yielding $O(N \cdot U)$ dot product computations, and b) sorting all the item scores in decreasing order for each user, yielding additional overall $O(U \cdot NlogN)$ computations in which U is the number of users and N is the number of items. One can decrease the retrieval time for the top-N items by limiting the number of dot product computations, however doing so may reduce the accuracy of the recommendations, because of the possibility that the excluded dot products produce higher scores than those which included in the computation. The task of delivering the top-N items from $\boldsymbol{p}$ and $\boldsymbol{q}$ is hence can be treated as an efficient search for the maximal dot product between a certain user $u \in \boldsymbol{p}$ and a set of items $qi \in \boldsymbol{q}$ as noted in Eq. (2).

$$u^T\boldsymbol{q} = \max \; u^T qi \;|_{\forall qi \in \mathbf{q}} \tag{2}$$

Several studies have focused on providing a solution to this search problem (Bachrach et al., 2014; Khosheneshin & Street, 2010; Koenigstein et al., 2012). The proposed solutions described in these studies try to achieve an appropriate balance between the accuracy of the top-N items and the corresponding retrieval time, thus, these solutions have sacrificed accuracy for the sake of speed in retrieving the top-N items. Instead, the proposed solution in this paper decreases the online retrieval time of the top-N items, increases the offline model building time, but do not sacrifice in accuracy.

The first solution (Khosheneshin & Street, 2010) for this retrieval problem suggested that instead of trying to search for the maximal dot product between users and items, one should search for the item vectors nearest the user vector. The proposed solution viewed the users and the items as a collection of vectors in a high dimensional space and searched for the closest item vectors for a given user within that space. Measuring the distance between vectors can be done by a number of methods, including calculating Euclidean distance between vectors. However, measuring the distance between two vectors entails more computation than applying the dot product of the vectors, and hence this may not offer a suitable solution to the problem at hand.

One way to decrease the number of dot product computations is by applying a data structure that encapsulates the latent features in such a way as to enable fast retrieval of the maximal (or near maximal) dot product (Koenigstein et al., 2012). They proposed representing the item factors in the form of a binary spatial-partitioning metric tree. After defining an upper bound for the dot product, their tree facilitates a branch and bound search approach. The acceleration achieved by this approach was not sufficient, because it was too expensive, in terms of computational resources, to be applied to all users. In light of this, they suggested that the tree be integrated with clustering over the users, ensuring that each user then relates to one cluster and for each cluster there are a limited number of item vector candidates. Once there is a need to deliver recommendations for a specific user, the system searches for the relevant cluster, acquires the item candidates from that cluster, and computes the distance between the user and only those item candidates. The ability of this method to pre-calculate and cache the nearest item vectors for each cluster, and the use of the tree to serve as a structure to quickly locate the cluster of an anonymous user effectively, reduces the number of computations that need to be applied for a given user. Although the researchers showed significant acceleration utilizing this combined method (tree + clustering), we believe there are limitations to this approach: a) information about the users is not always available in advance, and 2) clustering is often an expensive step that relies upon a similarity function to guide the learning process. The search for the maximal dot product among vectors was also explored under the assumption that the vectors in the corresponding matrixes are sparse (Ram & Gray, 2012). However, such an approach is irrelevant here because the SVD outputs fully dense vectors. Improving MF recommendations also been suggested in Ocepek, Rugelj, and Bosnić (2015). However, the study focus on improving the accuracy for users with few ratings and for items with few users which consumed them, namely the cold-start problem.

The latest relevant work in the literature presented a method for mapping the maximal dot product search to the Euclidian nearest neighbor search (Bachrach et al., 2014), and significantly improved the retrieval time with the use of a PCA-Tree. They first applied principle component analysis (PCA) on the output of the SVD. The PCA generates new ranked representation for the item latent features. After the PCA is applied and the importance of each new dimension is determined (since the PCA output ranked list of dimensions), the algorithm builds a tree data structure for the new item representation. The split of the items between the siblings of a tree node was performed according to the median value of the vector; this helps create a balanced binary tree and enables easy classification of a user to a leaf. Each leaf in the PCA-tree is comprised of the potential recommendations. The algorithm shows improvement in the time needed to generate a recommendations list, over the time is required by the approach of applying all the dot products. However, by using PCA to generate new important features the method may fail to fully utilize the predictive power of the original latent factors. Possibly reflecting this, the accuracy of the PCA tree was found to be inferior to the original SVD results.

A major difference between the solutions mentioned above and the suggested algorithm presented in this paper, is that the result of our offline algorithm improves both. The result of the algorithm enhances the accuracy of the original SVD results and reduces the retrieval time, while other solutions reduce the retrieval time at the expense of accuracy.

In this work we represent items in multiple compact decision trees – a forest – which are built offline. By doing so, we present a novel ensemble method for delivering top-N recommendations from a MF framework in real time without sacrificing accuracy. The use of decision trees to improve collaborative filtering recommendations has been suggested (Lee, 2010) but not from the results of a MF based method trained on rating data. In a sense, we continue work suggested previously (Bachrach et al, 2014) which also proposed representing the item factors in a tree. In contrast, we do not search for the nearest neighbors in that tree or apply any form of PCA to the latent features. Rather, we build numerous compact trees, each of which handles a small number of the problem dimensions and contains item candidates in its leaves. The learning process of each tree in the forest includes optimization of the dot product from Eq. (1) on a training dataset that comprises user vectors as well, thus maximizing the accuracy of the recommendations. Once there is a need to provide top-N items online to a given user, the user vector is classified quickly to one leaf in each tree where item vectors that will produce high dot products with the given user vector exist.

3. The ensemble method

We now present an overview of our suggested ensemble method for fast retrieval of top-N items from the SVD results. We start with a description and evaluation of a single weak recommendation model, and finalize with an overview on the building of the ensemble. Our ensemble is homogenous, built from multiple weak recommendation models, each of which is based on decision tree. The performance evaluation of the entire suggested ensemble algorithm, as well as a comparison to other algorithms, is presented in Chapter 4.

3.1. Learning and evaluating weak recommendations model

3.1.1. Building a weak recommendation model

Based on our observation of the dot product in the prediction Eq. (1) we concluded that achieving the maximum value of the dot product between the user and item vectors depends on the following conditions - (1) the values of the vectors should be as large as possible, and (2) in both of the vectors that are involved with the dot product, the correlated dimension values will be either positive or negative. These two conditions cause the dot product score result in a large positive value. We suggest a decision tree to classify the users and the items to a leaves, so that the dot product between the users and the items which are in the same leaves will result in a large positive value. This decision tree based approach for multiplying the items against the users' factors is described below.

1. Build a binary tree structure that holds all of the possible combinations of the plus and minus signs over the item factor values. This will lead to a binary tree with 2^{f-1} leaves where f is the dimension of an item (the number of factors). In practice the number of leaves is much smaller than 2^{f-1}, because we stop the splitting when the number of items in a node reaches a specific threshold.
2. The items are classified to the leaves according to the signs in their factors. Each leaf contains the items with the corresponding sequence of signs found in the factors. Items in the leaves are sorted in descending order based on their norm value.
3. When there is a need to deliver the top-N recommendations to a given user we classify the user to a leaf as follows:
   a. Start from the root of the tree, always choosing the node that holds the same sign as the sign in the user vector for the corresponding node/factor. This search involves f comparisons.
   b. Once you reach a leaf, apply the dot product between the user and the items in the leaf and deliver the top-N items with the highest score.

Fig. 1 illustrates a decision tree based on a sequence of signs with three factors, namely 1, 2, and, 3.

The items will be classified to the leaves according to their signs in the corresponding factor. For example, if we have an item whose vector is [2.5, 3.6, 4.5] it will be related to the leftmost leaf in the tree in Fig. 1 as the sequence of signs is [+++]. Recommendations for a user with the following vector [−2.3, −3.5, −5], will be obtained from the rightmost leaf in the tree in Fig. 1, since the sequence of signs is [− − −].

In a typical decision tree the most important feature is found in the root of the tree, and the least important features are located at the bottom of the tree. Accordingly, in our item factors decision tree, the most important factor should be in the root node, and the least important factors in the nodes one level before the leaves. In order to obtain that, we evaluate the contribution of each factor on each level of the tree during the learning process and then decide on the splitting factor. Each factor splits the items differently and generates a different precision score over the training data. This precision is obtained using the training data, and we discuss later on the how to compute it, but it simply a quality score for the factor. A high score signifies high importance of the corresponding factor, and therefore the factor which gains the highest precision in a node will serve as splitting criterion for that specific node. Once

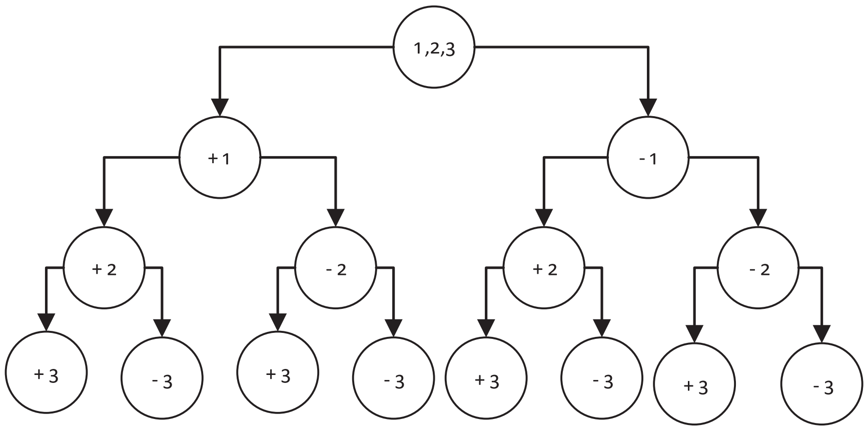


Fig. 1. A tree based on a sequence of signs with three factors in which factor 1 is the most important and 3 is the least.

Algorithm 1. SplitNode.

```
Input: Node V that has to be split, Factor f which determine the splitting criteria
Output: The node V gets assigned with left son and right son as a result of the split
(1) for each v'ϵ V.itemfactors //v' is an item vector
(2)    If v'[f]≥0 then //the value of the f dimension in the given item vector is positive
(3)       V.left.itemFactors ← v' //assign this item vector to the left son
(4)    else
(5)       V.right.itemFactors ← v'
(6)    end if
(7) end for
(8) for each u'ϵ V.userFactors
(9)    if u'[f]≥0 then //the value of the f dimension in the given user vector is positive
(10)       V.left.userFactors ← u' //assign this user vector to the left son
(11)    else
(12)       V.right.userFactors ← u'
(13)    end If
(14) end For
(15) V.score ← computePrecision(V) //compute precision for siblings and apply weighted average precision for getting the entire V's score
```

we assign a factor to one of the nodes, we eliminate this factor in the lower levels of the tree. For instance, assume that we are in the root of the tree, we have the item and user factors, and the number of dimensions (factors) is five. We evaluate the split of the root according to each one of the five factors, and each factor will split the items to the left and right child differently. In order to evaluate the score of the split/factor we also classify the user vectors to the left and right child according to the sign of the value of the exact factor itself. Thus, in the first step we end up with five trees, each with a root and two siblings representing one of our five factors. For each tree we calculate the precision (i.e., at 5) for the users in the leaves by providing recommendations only from within the items that exist in their leaf. The number of the users in the right child may not be the same as in the left child. Therefore, we normalize the score of each leaf by the number of users in it and calculate the sum together with the score from the other leaf to obtain the factor's overall precision score. At the end of that process we have five trees, each of which represent a factor and have a score. We now choose the tree that obtained the highest precision score and eliminate the other four trees. This process is repeated recursively for each node in the tree using the remaining items and users in the node. Once the number of items in a given node reaches a specific threshold or there are no user vectors left in a node, the learning process stops, and the node becomes a leaf. The tree is built effectively by optimizing the precision over the training data. This is our weak recommendation model because we use only a small number of factors in each tree.

Algorithm 1 presents the process for the split node method, and Algorithm 2 presents the building of a complete decision tree optimized by signs and factors.

Algorithm 1 accepts a node (comprising item and user factors) along with the examined factor, applies a split on the given node according to the factor, and compute a score (precision) for the split by using the 'computePrecision' function (line 11). This function calculates the precision resulting from the splitting which has been done according to the given factor over the training data. Once the items are classified to the siblings based on the factor, the users are also classified to the siblings in the same way, enabling us to measure the precision in each sibling and therefore for the entire split. Algorithm 2 is a recursive method for building a decision tree, calling Algorithm 1 whenever a node should be split. During the building process, Algorithm 2 chooses the factors that provide the highest precision as splitting factors.

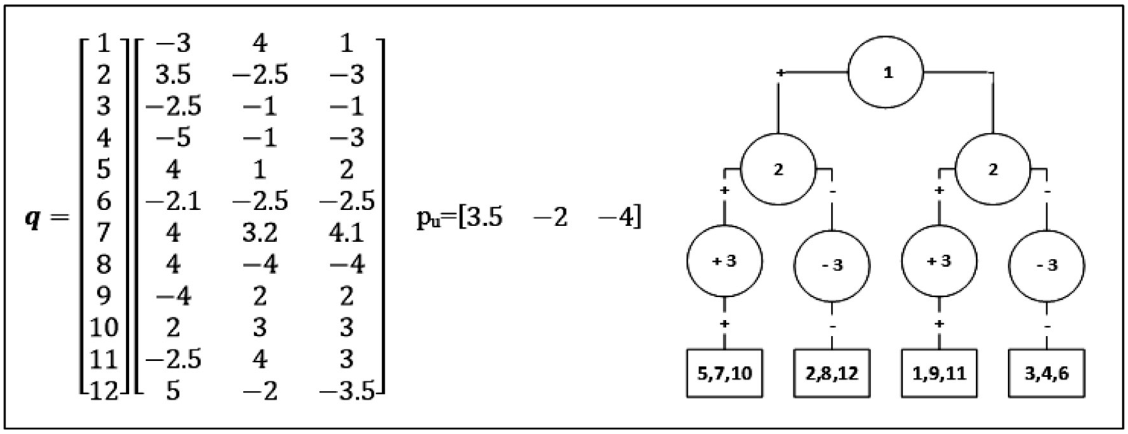
Since we learn this decision tree and optimize it based on the precision resulting from the dot products, we expect that each user will receive top-N items with a very high dot product score. Once we decide on the factor that determines the splitting of a node, we use the sign (negative or positive) of the actual value in the vector for classifying the item to one of the two siblings. We examined other splitting criteria besides the signs, such as the median of the factor values and the average of these values, but we found that the signs work best in our case.

3.1.2. Delivering top-N recommendations from a weak recommendation model

We now illustrate how to obtain the top-N recommendations for a given user from the tree we built in the previous section. Fig. 2 presents an example of items latent features matrix $\mathbf{q}$, a user latent features $\mathbf{p}_u$, and the tree that was constructed based on the process described in the previous section. Each node in the tree holds the dimension/factor which determines the split between the

Algorithm 2. BuildTree.

```
Input: A root node V first comprising all the factors, Set of factors F
Output: The node V representing a decision tree where each leaf comprising a set of factors
  (1) if |V.itemFactors| ≤ Threshold then //if a node have too few items we don't split
  (2)    return V
  (3) end If
  (4) Node winner
  (5) winner.score ← 0
  (6) for each f'∊F //for each available factor we examine its precision
  (7)    V.factor ← f'
  (8)    SplitNode(V,f') //ALGORITHM 1
  (9)    if V.score ≥ winner.score
  (10)        winner ← V //hooding the factor that leads to the highest score
  (11)    end if
  (12) end for
  (13) Delete winner.factor from F
  (14) V.factor ← winner.factor //the winner factor becomes the splitting criteria
  (15) BuildTree(winner.left,F)
  (16) BuildTree(winner.right,F)
  (17) Return V
```



$$\boldsymbol{q} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \end{bmatrix} \begin{bmatrix} -3 & 4 & 1 \\ 3.5 & -2.5 & -3 \\ -2.5 & -1 & -1 \\ -5 & -1 & -3 \\ 4 & 1 & 2 \\ -2.1 & -2.5 & -2.5 \\ 4 & 3.2 & 4.1 \\ 4 & -4 & -4 \\ -4 & 2 & 2 \\ 2 & 3 & 3 \\ -2.5 & 4 & 3 \\ 5 & -2 & -3.5 \end{bmatrix} \quad p_u=[3.5 \quad -2 \quad -4]$$

Fig. 2. Example of item latent features matrix, user latent features, and the resulting weak recommendation model built from 3 factors. The resulted latent factor matrices coming from the SVD will have more factors.

siblings. The leaves of the tree also hold the items from **q**. The items were classified to the leaves according to the signs of the related dimension which are denoted inside the nodes. The first column in **q** stands for the item IDs. For example, item 6 is classified to the rightmost leaf, because its sequence of signs in dimensions 1, 2, 3 is [− − −] respectively.

Once there is a need to deliver top-N recommendations to the user u, we classify the user to one of the leaves in the tree according to the signs of the corresponding factor. In the example presented in Fig. 2, user u has a positive value in factor 1 and negative values in factors 2 and 3; therefore the user will get recommendations from the leaf which holds items 2, 8, and 12. Assuming we need to deliver only the top 2 items, we apply the dot product between the vector of user u and the vectors of items 2, 8, and 12. This dot product yields the following scores: Score (item_2) = 5.4, Score (item_8) = 6.16, and Score (item_12) =5.95. As we need to deliver only two of the items, we choose the two items with the highest score, i.e., 8 and 12.

To evaluate the suggested tree described in Section 3.1.1 we perform a brief experiment and compare the top-N items generated from the tree to the top-N items resulting from a public version of SVD called SVDPlusPlusFactorizer taken from the Apache Mahout library[1] which we will use as a baseline. Since the task is to deliver top-N recommendations we used *precision@5* as an accuracy measure, training the model and computing item predictions for each user, and then choosing the five highest predictions for each user. Precision is defined as the ratio of relevant recommended items (r) to number of items presented (in our case $n = 5$), shown in Eq. (3) (Herlocker, Konstan, Terveen, & Riedl, 2004).

$$\text{Precision at } n = \frac{r}{n} \tag{3}$$

Here r is the number of items which were relevant to the user, measured against the hidden test set of each user, and n is the size of the presented list. $r \le n$. Precision represents the probability that a selected item is relevant. This measure is adapted from the information retrieval (IR) area where usually Recall is also been measured. Recall is similar to Precision but represents the probability that a relevant items will be selected. While knowing the relevance of each document to a given query in IR may be possible, in RS it will require that we will know on each item if its relevant for each user, which is of course impossible. Therefore measuring recall is impractical and do not reflect the accuracy of a recommender. We hence do not present the Recall results in this paper.

We use the baseline to computes predictions for all of the items for each user, while the tree method computes predictions only for the items that are in the leaf that the user is classified to. The MovieLens 1 million dataset was divided into a 70% training set and a 30% testing set. We build the baseline model on the training set and measure *precision@5* on the testing set. The latent factors resulting from the baseline served as the input for the building process of the tree model. Since the tree is aimed to serve as

[1] http://mahout.apache.org/.

Table 1
Precision at 5 for the MovieLens 1 million dataset - 32 factors – weak recommendation model.

	SVDPlusPlusFactorizer - Mahout	Tree – 32 factors	Tree – 16 factors
Precision at 5	0.114	0.0886	0.074
Retrieval time	4.4 s	0.75 s	0.5 s

a weak recommendation model we also trained it with only half of the factors, thus, we had two version of the tree, one with all the factors and another with only half of the factors. Once the trees were built, we generate five recommendations for each user and measure the precision over the test data. Table 1 presents the results of this experiment. Note that the results of the baseline serve as a gold standard. The SVD related parameters - number of factors, number of iterations, learning rate, and the regularization term were optimized for the baseline using trial and error. The number of possible leaves in the trees is 2^{f-1}, where f is the number of factors. In this experiment we use 32 factors thus potentially we have 4.2950e+9 which is impractical. We stop the splitting of a node and going down the three whenever the number of items in it reach 300. Here we choose 300 as the size of a leaf using trial and error but we later present results with different values for this parameter. When the minimum size of a leaf is 300 and the number of items is 3706 (MovieLens 1 million dataset), it creates $\sim$13 leaves. In practice it may even be smaller because we stop the splitting of a node also when there are no users with corresponding sequence of signs.

The baseline achieves better accuracy compared to the tree approaches. This could be due to the fact that some of the user vectors weren't multiplied against the item vectors that produce the highest possible dot product score. For example, if the vectors of the user and the item both have low norms it will result in a very low dot product, while in some cases user and item vectors with a sequence of signs that have high values in certain dimensions may lead to a very high dot product. Hence, we need to compensate for the errors that such a single tree may yield. The tree based on 16 factors achieves reasonable accuracy in compared to the baseline which is a strong recommendation model (Dietterich, 2000), much better than a random, thus may serve as a proper weak recommendation model for an ensemble approach.

The results show that acquiring top-N recommendations from the trees is much faster than applying all of the dot products on the users and the items. It should be emphasized that we compare the time took for fetching recommendations from the models and not the building time. The building time of a tree may be very long in certain settings even longer than applying all the dot products.

3.2. Building an ensemble of recommendation models to acquire top-N items

To compensate for the potential error(s) caused by a single tree, we suggest building a group of trees, a forest, in which the majority of the trees in the forest deliver items with a high dot product for a single user in any case. This is based on the assumption that if a single tree, or even more than one tree, erroneously assigns bad items to a user, the remaining trees will assign good items and compensate for the error caused by the single tree. This is based on the fact that under certain controlled conditions, the aggregation of information from several weak learners, resulting in accuracy that is often superior to the one that can been made by a single strong learner (Dietterich, 2000).

Algorithm 3. BuildForest.

Input: The factors F, the proportion p of the factors that will be included in each tree, the number of trees a that each factor should be appeared in, and the sl minimum number of items in each leaf

Output: The *forest* - a set of trees

```
(1) FactorGroups groups←getFactorGroups(f.size,p,a)
(2) TreeList forest←getEmptyListForTrees()
(3) for g ϵ groups
(4)     BinaryTree tree←getEmptyBinaryTree()
(5)     tree.userFactors←getUserSubSpace(F,g)
(6)     tree.itemFactors←getItemSubSpace(F,g)
(7)     BuildTree(tree,g) //Algorithm 2
(8)     forest.add(tree)
(9) End for
(10) Return forest
```

An important condition for the success of an ensemble is the diversity among the weak learners. We need to identify what makes each tree in our ensemble unique, or more specifically, how each tree is different from the others. To fulfill that, we suggest that each tree consider and treat only a portion of the factors that assigned to the tree randomly, and subsequently each factor will be considered by only a portion of the trees in the ensemble. Table 2 presents a set of parameters used to define our ensemble of decision trees algorithm for retrieving the top-N items from the SVD results.

The number of factors f is already defined by the SVD output. Hence, we determine only the number of trees that each factor will be included (a) in and the portion of the factors that will be included in each tree (p). For example, assume the following: we have 32 ($f = 32$) factors, we wish to include 50% ($p = 0.5$) of the factors in each tree, and each factor will appear in at least three trees ($a = 3$). In this setting we will have six trees in the forest ($3/0.5 = 6$), and each tree will include 16 factors ($32 \times 0.5 = 16$). This is how we control the diversity among the trees and the number of trees in the ensemble. The size of a leaf (sl) defines the tree depth. A low value for sl indicates a very deep tree with only sl dot product computations for each user in each tree, and in practice, this improves the retrieval time, since there are less dot products to calculate for each user. However, since the user is classified to a leaf with only a few potential items, the probability that these specific items will create a maximal dot product is low. Algorithm 3 illustrates the steps to build the entire forest. The BuildTree call in line 7 is the call to Algorithm 2 to build a single tree. This call also encapsulates multiple calls to Algorithm 1 whenever splitting a node is required.

Algorithm 3 accepts four parameters: the user and item factors resulting from the SVD (encapsulated in the parameter F), the

Table 2
Parameters of the forest.

Parameter name	Notation	Details	Comments
Factors	f	Total number of factors	Defined as a parameter to the SVD
Percent	p	The portion of the factors that will be included in each tree	Value between 0.1 and 1
Appearance	a	The number of trees that each factor will be included in	A good value will be to include each factor in half of the trees
Tree factors	ft	The number of factors in each tree	$ft = p * f$
Number of trees	nt	The number of trees in the ensemble	$nt = a/p$
Leaf size	sl	The minimum number of items in a leaf	Determine the tree depth

portion of the factors that will be included in each tree (p), the number of trees that each factor should be included in(a), and the minimum number of items in a leaf (sl). The result of the algorithm is defined in line 2, the parameter 'forest', which is a list of decision trees populated in line 8.

The function 'getFactorGroups' in line 1 of Algorithm 3 accepts the total number of factors ($f.size$), the number of trees (a) that each factor should appear in, and the percent (p) of the factors that each tree should cover. The last two parameters together define the number of trees ($nt = a/p$) in the forest and the function itself returns a list of groups in which each group is comprised of a set of factors. The algorithm then builds an optimized tree for each group (line 7) as described in the previous section and adds it to the forest list (line 8). Once the forest is complete, we end up with a unique set of trees, each with the same number of factors. Each tree is comprised of a different combination of factors, and each factor exists in several trees. This protocol creates a set of highly diverse trees, each is independent from the other trees in the forest and can deliver autonomous recommendations to any user.

When there is a need to deliver recommendations for a given user vector, we obtain recommendations from each of the trees as described in Section 3.1, and then we aggregate the results from all of the trees to generate the final list of recommendations. For example, assume we have two trees in our forest and we need to deliver the top two items. The first tree delivers the list of recommendations LR = [10:5.6|20:3.3|30:2.2], and the second tree delivers LR = [40:4.2|30:3.1|20:3]. In this case the final list of recommendations will be LR = [20:6.3|10:5.6]; thus items 20 and 10 will be recommended. We total all the scores for each item in all of the trees to compute its final score.

4. Experimental evaluation

In order to evaluate the performance of the proposed algorithm, we compare it with the algorithm suggested by Bachrach et al. (2014). From now on we refer to this algorithm as PCAMedian and to our algorithm as ForestSigns. The input for both algorithms consists of the item and user factors resulting from the SVD. In order to get this information from the SVD we trained the SVDPlusPlusFactorizer[2] and stored the factors for later usage in the ForestSigns and PCAMedian algorithms. We also measured the precision of the SVDPlusPlusFactorizer to deliver top-N items and the retrieval time (the time it took to compute the top-N recommendations). From now on we refer to the SVDPlusPlusFactorizer results (precision and time) as the baseline. It should be noted that the baseline method is not restricted by time and computes all the possible dot products among the users and the items once the learning process of the factors over the training set has been performed.

We then trained the PCAMedian and ForestSigns algorithms using the factors generated by the baseline in order to produce the structure that enables fast retrieval of the top-N items. The baseline approach is considered the gold standard since it applies all of the dot product computations and selects the best top-N recommendations, while the PCAMedian and the ForestSigns algorithms apply only a subset of the dot products. Therefore, one does not expect to receive better accuracy from either the PCAMedian or the ForestSigns algorithm but rather expects to see a trade-off between retrieval time and accuracy in which major improvement in the retrieval time will be associated with a slight downgrade in the quality of recommendations.

[2] http://mahout.apache.org/.

Table 3
The benchmark datasets used in the experiments.

NAME	#EVENTS	# ITEMS	# USERS	SPARSITY
MOVIELENS 1 MILLION	1000,000	3706	6040	0.9553
MOVIELENS 10 MILLION	10,000,811	10677	69,878	0.999
YAHOO!	354,000	1000	15,400	0.977

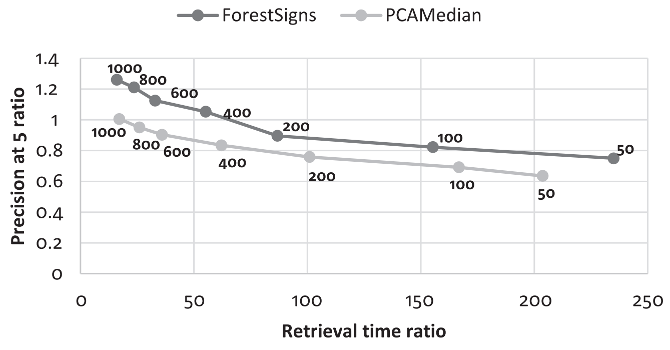


Fig. 3. Results for the MovieLens 10 million dataset. The labels next to the points indicating the size of the leaf in the corresponding run.

4.1. Experimental protocol

We conducted experiments on three datasets: MovieLens[3] 1 million ratings, MovieLens 10 million ratings, and Yahoo! Music ratings for user-selected and randomly selected songs version 1.0 dataset which is available through the Yahoo! Webscope data sharing program.[4] Table 3 presents some statistics about the datasets.

For each of the datasets we recommend items that the user might rate high given other movies he or she has rated. To model this scenario we compute the rating that the user will give an item, and we choose the top-N items which obtain the highest scores for the given user. The top-N items will always be items that the user has not yet seen. Each user's consumption set is divided into a training and a testing set. For each user u, we randomly pick a number k between 0 and the number of items that u consumed. We then select k random items from u's consumption set, using them as the training set; the remaining items serve as the testing set. We execute all algorithms on the three datasets. The training set is used to build the baseline, and the factors generated from the baseline are used as input to the PCAMedian and ForestSigns algorithms. Compared to a standard 5-fold cross validation 90% train 10% test, this protocol hides more data per user and we believe it's reflecting more accurately a real system than hiding 10% of the ratings for all the users and train for the remaining 90%. As the task is to provide top-N recommendations, we found *precision@N* (Herlocker et al., 2004; Shani & Gunawardana, 2011) to be an appropriate metric for evaluating model performance. We provide results for $N = 5$, i.e., where the algorithm computes five recommended items. We also experimented with different values of N (1 and 10) but found no sensitivity to the size of the recommendations list. We ran the algorithms five times, and report the average results. All algorithms use the same train-test split.

4.2. Results

Figs. 3–5 present the results of the algorithms with relation to the baseline for the three datasets. The x axis depicts the ratio between the retrieval time of the examined algorithm and the retrieval time of the baseline algorithm (the time it took the baseline

[3] http://www.grouplens.org/node/73.

[4] http://webscope.sandbox.yahoo.com/catalog.php?datatype=r.

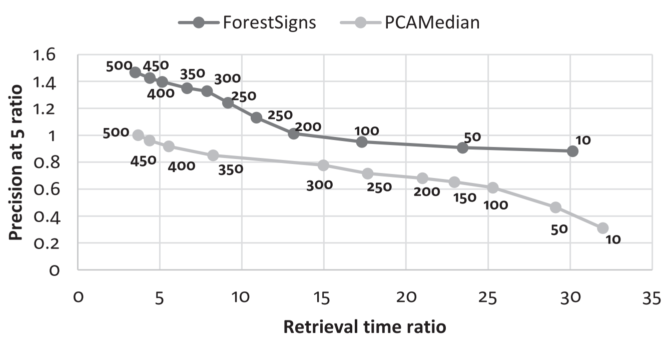


Fig. 4. Results for the MovieLens 1 million dataset.

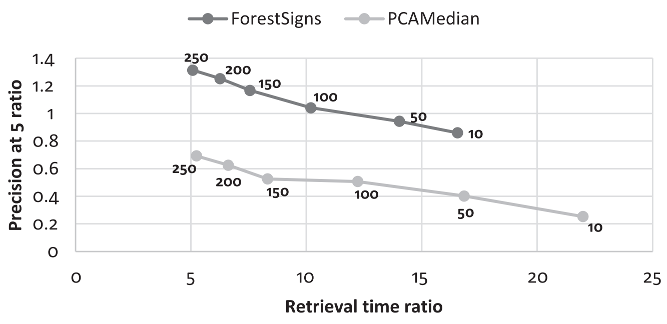


Fig. 5. ForestSigns results for the Yahoo! music dataset.

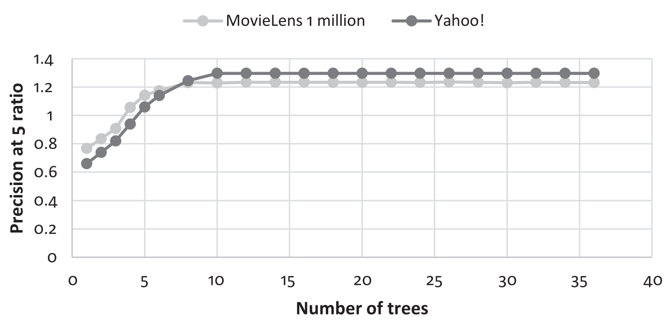


Fig. 6. The effect of the number of trees on precision for the MovieLens 1 million dataset.

algorithm to deliver top-N items); therefore the x axis represents the acceleration of the algorithms over the baseline. The y axis illustrates the precision at 5 ratio for the examined algorithm and the baseline. For instance the point (100, 0.8) of the PCAMedian graph in Fig. 3 indicates that the PCAMedian is 100 times faster than the baseline at retrieving the top-N items while obtaining 80% of its accuracy when we set the number of items in a leaf to 200. Each point in the curves indicating the performance of the algorithm with different number of items in a leaf. The number of items in a leaf affecting directly on the retrieval time as well as on the accuracy. Thus each point in each one of the plots in the graphs stands for a complete run of the algorithm with a different number of items in a leaf. This number is presented in the labels next to the points in the graphs. Note that Yahoo! dataset comprises only 1000 items and hence we didn't test it with size of a leaf greater than 250 since it would have been create a very small trees. In fact, choosing size leaf of 250 items for the Yahoo! dataset will generate a tree with 4 leaves on average and this implies that only 2–3 factors may be involved in it. In the results presented in Figs. 3–5 we set the number of trees to six (by setting $a = 3$ and $p = 0.5$ – see Table 2 above).

We expect that low retrieval time will reduce the accuracy, and very high retrieval time, if this occurs, will cause the accuracy of the ForestSigns and PCAMedian to converge to the accuracy of the baseline. We can see from the results that the highest retrieval time was obtained in the MovieLens ten million dataset (largest), and the lowest was obtained in the Yahoo! dataset (smallest). The ForestSigns algorithm built smaller trees than the PCAMedian since each tree treats only a portion of the factors; in the experiments described here we include only 50% of the factors in each tree which creates smaller dot products and reduce the retrieval time. Thus, although the retrieval is handled via multiple trees in the ForestSigns method, these trees are less deep than the PCAMedian tree, ensuring competitive retrieval time. On the other hand, the accuracy is higher in the ForestSigns method in all three datasets because of its ability to benefit from ensemble characteristics. It is well know that good ensembles (which keep high diversity among the weak learners) achieves higher accuracy than a weak learner, in our case a single tree, as well as better accuracy than a strong learner, in our case the baseline SVD and the PCAMedian. There are evidences for up to ~25% (Freund & Schapire, 1996) improvement by using ensemble method over a single strong learner. This is the reason of why we see in the results a precision ratio which higher than 1 – these are cases where the suggested ensemble achieve higher accuracy than the baseline SVD. As expected, the PCAMedian algorithm never accomplished this, because its precision is never better than the baseline and hence the ratio is always less than 1. In conclusion, the reasons that our method provides higher accuracy in comparison to the PCAMedian and to the baseline SVD is due to two aspects – (1) our method is an ensemble of trees known to improve the accuracy of a single tree, and (2) the learning process of each tree in the ForestSigns is optimized for maximizing the precision.

Based on the results presented in Fernández-Delgado, Cernadas, Barro, and Amorim (2014) we also tried the classic random forest strategy to generate the forest, but the accuracy and retrieval time were inferior to our suggested method. For example in the MovieLens one million dataset the classic version of random forest obtains a precision at 5 ratio of 0.73 and a retrieval time ratio of 7. Some possible explanations are: (1) random forest creates deep trees that minimally decrease the computational cost of the retrieval process but greatly reduce the accuracy, and (2) random forest requires a relatively large number of trees to obtain good predictive performance, which means an increase in the retrieval time because the use of many trees in the online voting phase.

4.3. The effect of number of trees in the forest

We now wish to examine the effect of the number of trees in the ForestSigns algorithm on the results. The number of trees in Figs. 3–5 was set to 6 (by setting $a = 3$ and $p = 0.5$ – see Table 2 above), however, this parameter may affect not only the retrieval time but also the accuracy. Fig. 6 presents our findings for the MovieLens one million and for the Yahoo! datasets when we varied the number of trees in the forest while fixing the size of a leaf to 150.

The results show that only a small number of trees is needed to obtain maximum improvement in accuracy. At some point, adding more trees to the forest does not improve precision or harm it.As we can see in Fig. 6, increasing the number of trees beyond 10, causes the precision ratio to stay the same. It should be noted however that adding trees to the forest increase the retrieval time.

4.4. The memory and the computation costs for training the models

In the previous sections we show that the ForestSigns algorithm can deliver the top-N recommendations very fast and more accurately in comparison to other state-of-the-art methods. We now

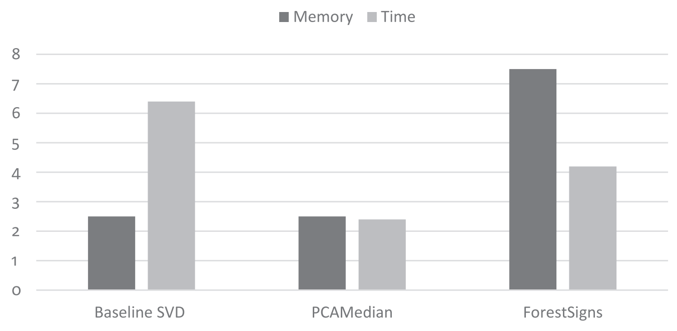


Fig. 7. The training cost and memory footprint of each model.

wish to describe the computational and memory costs of building and maintaining the ForestSigns model. The computational cost and the memory footprint of the ForestSigns is directly depends on the baseline SVD which generates the user and item factors since these factors are the input to the ForestSigns as well as to the PCAMedian. Let's consider the case of the MovieLens 1 million dataset. In this dataset we have 3706 items, 6040 users and if we will train the baseline SVD with 32 factors we will have a model's size of $\sim$2.5 MB ($3706 \times 32 \times 8 + 6040 \times 32 \times 8$) assuming each value in the factors matrix is 8 bytes. The PCAMedian holding exactly the same factors, hence its size is the same. The ForestSigns with 6 trees where each factor will be exists in half of the trees will have a model's size 3 times the size of the baseline. Note that in practice the memory requirements are larger (overhead) and subject to implementation. The training time for the baseline SVD with 32 factors and 50 iterations, took 19 min. The result of the baseline is the input to the PCAMedian and to the ForestSigns, hence this training time is omitted from Fig. 7. Building the PCAMedian and the Forest signs took 2.5 and 4.2 s respectively with 65% of the data as training set. Fig. 7 shows the computational cost for training the models and their resulted memory footprint once we have the latent factors from the baseline SVD. Note that it might be that ForestSigns training time may be very long and even longer than applying all the possible dot products in the baseline especially in case the number of factors is large. However, once it has been build (and the suggestion of course to build it offline), it will provide more accurate and faster retrieval of recommendations for a given user.

We have implemented the algorithms in Java and run the experiments over a Dell Latitude E6530 machine, equipped with Intel Core i-7-3720Qm CPU @ 2.6 GHz, and 8GB of RAM. It's a 64 bit machine with windows 7.

5. Conclusions and future research

Matrix factorization based models have demonstrated high predictive accuracy as well as good scalability. However, once the factors have been learned, there is a computational bottleneck in delivering recommendations online. Furthermore, pre-computing recommendations for all users in a MF framework is an inadequate solution, because it isn't possible to integrate the most up-to-date user information into the recommendations list.

In this paper we present a novel ensemble method based on a forest of decision trees designed to overcome these problems. Each tree in the forest is built and optimized offline during a learning process which maximizes the precision of the top-N recommendations. We then suggest to use the forest online to deliver top-N recommendations for a given updated user vector. We compare the performance of the suggested method to the best known algorithm addressing the problem, the PCAMedian, and to the baseline SVD. The experimental results demonstrate that our method generates higher quality recommendations than the PCAMedian and the baseline SVD, while consuming competitive retrieval time to PCAMedian. Once there is a need to deliver top-N items to a given user, the user vector is classified to one leaf in each tree and multiplied against all the items in these leaves. The result of this process is a diverse and accurate top-N recommendations list. Since the ForestSigns method only contains the item vectors in the model, it allows the user vector to be changed frequently, thereby integrating recent user activities in the recommendations. Sensitivity analysis shows that it is not necessary to include a large number of trees in the forest or a large number of items per leaf, leading to a relatively compact ensemble model for quickly retrieving high quality recommendations. The suggested method makes the SVD more practical in real life situations which often comprise large numbers of users and items. The only drawback the suggested method have is the extra training time that is done offline for constructing the forest once the factors have been learned in the base SVD.

Future research may focus on building an item-based model using the item latent features resulting from the SVD to avoid the dot product computations online. The item-item similarities based on the latent features will be computed offline, while the top-N items for the user will be computed online. Such an approach is utilizing the predictive power of the SVD and still delivering top-N items quickly. The top-N items for a given user will be extracted from the item-based model as the items that are most similar to the items the user has already consumed. In any case, such an approach is subject to the fact the item-based model will obtain satisfactory accuracy.

References

Bachrach, Y., Finkelstein, Y., Gilad-Bachrach, R., Katzir, L., Koenigstein, N., Nice, N., et al. (2014, October). Speeding up the Xbox recommender system using a euclidean transformation for inner-product spaces. In *Proceedings of the 8th ACM.*

Bennett, J., & Lanning, S. (2007, August). The netflix prize. In *Proceedings of KDD cup and workshop: 2007* (p. 35).

Cremonesi, P., Koren, Y., & Turrin, R. (2010). Performance of recommender algorithms on top-n recommendation tasks. In *Proceedings of the fourth ACM conference on recommender systems* (pp. 39–46). ACM.

Dietterich, T. G. (2000, June). Ensemble methods in machine learning. In *International workshop on multiple classifier systems* (pp. 1–15). Berlin Heidelberg: Springer.

Fernández-Delgado, M., Cernadas, E., Barro, S., & Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research, 15*(1), 3133–3181.

Freund, Yoav, & Schapire, Robert E. (1996). Experiments with a new boosting algorithm. In *icml: Vol. 96* (pp. 148–156).

Herlocker, J. L., Konstan, J. A., Terveen, L. G., & Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS), 22*(1), 5–53.

Karatzoglou, A., Amatriain, X., Baltrunas, L., & Oliver, N. (2010, September). Multiverse recommendation: N-dimensional tensor factorization for context-aware collaborative filtering. In *Proceedings of the fourth ACM conference on recommender systems* (pp. 79–86).

Khoshneshin, M., & Street, W. N. (2010, September). Collaborative filtering via euclidean embedding. In *Proceedings of the fourth ACM conference on recommender systems* (pp. 87–94). ACM.

Koenigstein, N., Ram, P., & Shavitt, Y. (2012, October). Efficient retrieval of recommendations in a matrix factorization framework. In *Proceedings of the 21st ACM international conference on Information and knowledge management* (pp. 535–544). ACM.

Koren, Y. (2008, August). Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 426–434). ACM.

Lee, S. L. (2010). Commodity recommendations of retail business based on decision tree induction. *Expert Systems with Applications, 37*(5), 3685–3694.

Ocepek, U., Rugelj, J., & Bosnić, Z. (2015). Improving matrix factorization recommendations for examples in cold start. *Expert Systems with Applications, 42*(19), 6784–6794.

Ram, P., & Gray, A. G. (2012, August). Maximum inner-product search using cone trees. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 931–939). ACM.

Shani, G., & Gunawardana, A. (2011). Evaluating recommendation systems. In *Recommender systems handbook* (pp. 257–297). Springer US.