Adaptive Bootstrapping of Recommender Systems Using Decision Trees

Nadav Golbandi
Yahoo! Labs, Haifa, Israel
nadavg@yahoo-inc.com

Yehuda Koren
Yahoo! Labs, Haifa, Israel
yehuda@yahoo-inc.com

Ronny Lempel
Yahoo! Labs, Haifa, Israel
rlempel@yahoo-inc.com

ABSTRACT

Recommender systems perform much better on users for which they have more information. This gives rise to a problem of satisfying users new to a system. The problem is even more acute considering that some of these hard to profile new users judge the unfamiliar system by its ability to immediately provide them with satisfying recommendations, and may quickly abandon the system when disappointed. Rapid profiling of new users by a recommender system is often achieved through a bootstrapping process - a kind of an initial interview - that elicits users to provide their opinions on certain carefully chosen items or categories. The elicitation process becomes particularly effective when adapted to users' responses, making best use of users' time by dynamically modifying the questions to improve the evolving profile. In particular, we advocate a specialized version of decision trees as the most appropriate tool for this task. We detail an efficient tree learning algorithm, specifically tailored to the unique properties of the problem. Several extensions to the tree construction are also introduced, which enhance the efficiency and utility of the method. We implemented our methods within a movie recommendation service. The experimental study delivered encouraging results, with the tree-based bootstrapping process significantly outperforming previous approaches.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms, Experimentation

Keywords

collaborative filtering, decision tree, new user, recommender systems, user cold start

1. INTRODUCTION

Modern consumers are inundated with choices. Electronic retailers and content providers offer huge inventories of products, with unprecedented opportunities to meet a variety of special needs and tastes. Matching consumers with the most appropriate products is not trivial, yet is key to enhancing user satisfaction and loyalty. This motivates the study of recommender systems, which analyze patterns of user interest in items or products to provide personalized recommendations of items that will suit a user's taste.

One particular challenge that recommender systems face is handling new users; this is known as the *user cold start problem.* New users are crucial for the recommendation environment, and providing them with a good initial experience is essential to growing the user base of the system. However, the quality of recommendations strongly depends on the amount of data gathered from the user, making it difficult to generate reasonable recommendations to users who are new to the system. In order to quantify this point, Fig. 1 shows how the error on Netflix test data decreases as users provide more ratings. Furthermore, pleasing new users is all the more challenging, as they often judge the system based on their first few experiences. This is particularly true for systems based on explicit user feedback, where users are required to actively provide ratings to the system in order to get useful suggestions. The essence of bootstrapping a recommender system is to promote this interaction, encouraging users to invest in a low-effort initial dialog that leads to an immediately rewarding experience.

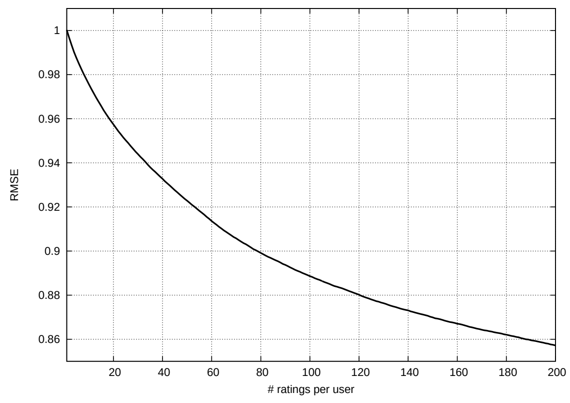


Figure 1: The test error rate vs. number of train ratings per user on the Netflix data. Lower y-axis values represent more accurate predictions. The x-axis describes the exact number of ratings taken for each user. For a certain x value, we are considering only users that gave at least x ratings. For each such user, we sort the ratings in chronological order and take the first x ratings into account. Results are computed by the factorized item-item model [11].

This paper introduces a method for eliciting information from new users in a manner leading to high quality recommendations, while minimizing user effort in the process. The method involves an interview process, where users are asked for their opinions on certain deliberately chosen products. In order to achieve better accuracy and user experience, the interview process adapts to user responses, such that the answer to a question influences the system's choice of the following question.

Our main contribution is an adaptive bootstrapping algorithm based on optimizing a formally defined cost function. The core of the method is an efficient decision-tree-based recommender algorithm, suitable for an adaptive bootstrap process. While decision trees are well recognized for general classification tasks, they have yet to play a significant role within recommender systems. We show that when well crafted for the task, they significantly outperform previous approaches to eliciting user feedback.

The reader is invited to experience a demo of this work, involving a bootstrap movie recommender, at: `http://collab02.yrl.re3.yahoo.net/`.

2. NOTATION AND DATASET

We are given ratings for m users within user set $\mathcal{U} = \{1, \ldots, m\}$ and n items within item set $\mathcal{I} = \{1, \ldots, n\}$. We reserve special indexing letters to distinguish users from items: for users u, v, and for items i, j. A rating r_{ui} indicates the preference by user u of item i, where high values mean stronger preference. For example, values can be integers ranging from 1 (star) indicating no interest to 5 (stars) indicating a strong interest. Usually the vast majority of ratings are unknown. For example, 99% of the possible ratings are missing in the Netflix data because a user typically rates only a small portion of the movies. The (u, i) pairs for which r_{ui} is known are stored in the set $\mathcal{R} = \{(u, i) \mid r_{ui} \text{ is known}\}$, which is known as the *training set*. We distinguish predicted ratings from known ones, by using the notation $\hat{r}_{ui}$ for the predicted value of r_{ui}. The set of items rated by user u (in training set) is denoted by $\mathrm{R}(u)$. Similarly, $\mathrm{R}(i)$ stands for the set of users rating item i. A test set, used to evaluate prediction accuracy, is denoted by $\mathcal{T}$.

Our test bed is a large movie rating dataset released by Netflix as the basis of a well publicized competition [2]. The dataset contains more than 100 million date-stamped ratings performed by about 480,000 anonymous Netflix customers on 17,770 movies between Nov 11, 1999 and Dec 31, 2005. Ratings are integers ranging between 1 and 5. To maintain compatibility with results published by others, we adopt some common standards. We evaluate our methods on a separate test set ($\mathcal{T}$) designed by Netflix, which contains over 2.8 million ratings (also known as the "Qualifying set"). The quality of the results is measured by their root mean squared error (RMSE) $\sqrt{\sum_{(u,i)\in\mathcal{T}} (r_{ui} - \hat{r}_{ui})^2 / |\mathcal{T}|}$.

3. RELATED WORK

A popular approach to building recommender systems is *Collaborative Filtering* (CF), a term coined by the developers of the first recommender system - Tapestry [6]. CF relies only on past user behavior, e.g., their previous transactions or product ratings. It analyzes relationships between users and interdependencies among products, in order to identify new user-item associations. A major appeal of CF is that it is domain free, yet it can address aspects of the data that are often elusive and difficult to profile.

The two primary areas of CF are *neighborhood methods* and *latent factor models*. Neighborhood methods compute the relationships between items or, alternatively, among users. The item-item approach [14, 18] evaluates the preference of a user to an item based on ratings of "neighboring" items by the same user. An item's neighbors are items that tend to be scored similarly when rated by the same user. Latent factor models are an alternative approach that explains ratings by characterizing both items and users on factors inferred from the pattern of ratings. One of the most successful realizations of latent factor models is based on *matrix factorization*, e.g., [12].

3.1 Providing recommendations to new users

Relatively few prior works deal with handling of users new to the recommender system. An early attempt was [10]. A comprehensive study was conducted by the GroupLens team [16], and was later enhanced [17]. We have recently published a work [5] on static seed set construction, a construction which is also the main theme of the other published works.

3.1.1 Bootstrapping using static seed sets

Learning the preferences of new users is efficiently achieved by a short interview during which they are asked to rate several carefully selected items of a seed set. One might intuitively expect the items in such a seed set to have certain properties, leading to various selection criteria which we discuss in the followings.

Popularity. A prime guideline is that the seed set should include familiar items, since asking users to rate obscure items is mostly futile. Thus, a reasonable strategy is to ask the user about the most popular items, i.e., those that have received the most ratings in the past [16].

Contention. Items in the seed set should not only be familiar to the users, but also indicative of their tendencies. After all, finding that a user liked an item that is also liked by everyone else, provides less insight than discovering that a user likes a more controversial item. Two common measures to quantify the contention associated with an item are the variance and entropy of its ratings [10, 16, 17]. For example, a seed set can be populated with the items of highest entropy. In the following, we refer to this method by *Entropy*.

Albeit, as identified in [16], maximizing a contention-related criterion with disregard to item popularity is unwise. There is little point in presenting highly controversial, yet lesser known items, which are highly indicative only with respect to a narrow group of users that are familiar with them. Furthermore, it has been claimed [16] that the widest spread in rating pattern (indicating contention) tends to be associated with those items receiving fewer ratings, while popular items are less controversial. In this sense, contention is negatively correlated with popularity.

Therefore, contention is only useful when combined with popularity. Among top performing methods in [17] are hybrids of popularity and entropy such as *Entropy0* and *HELF*.

Coverage. Items are useful when they possess predictive power on other items. Controversial items are not necessarily best at this. For example, within the Netflix movie rating dataset, most contention metrics identify *Napoleon Dynamite* as a highly controversial movie, exposing much disagreement among users. Yet, it is well known that ratings of Napoleon Dynamite are very weakly correlated with ratings of other movies. This is an example of a movie which, despite being ranked high on the contention and familiarity axes, is less useful within the seed set, as it is not very instructive on the perceived quality of other items. The coverage measure [5], emphasizes those items more heavily co-rated with other items.

Optimization. In our recent work [5], we proposed the *GreedyExtend* algorithm, which iteratively extends the seed set. In each iteration, an item is added in a way that minimizes the error rate when limiting the prediction algorithm to learn only interactions involving seed items. This leads to better performance than the aforementioned techniques of constructing the seed set.

3.1.2 *Bootstrapping by adaptive seed sets*

Rashid et al. [16] recognized the importance of adaptive interviewing processes. They call them "personalized strategies" in which items presented to a user are tailored to the so-far learned taste. Their approach is based on item-item personalization, where users are displayed items similar to those they have indicated to know. The rationale is to maximize the number of items familiar to the user. In practice, however, the result is a less diverse list, apparently leading the user to provide redundant feedback. Hence, in terms of accuracy, the method lags behind fully static methods such as *Popularity* and *HELF* [16].

A later work by that group [17] contemplates using decision trees for achieving an adaptive interview process. They propose to first cluster users into groups, and then to build a classical decision tree classifier such as ID3 that will closely recover the known user clusters based on user ratings of specific items. However, the authors dismiss the idea as not practically feasible, and instead suggest an adaptive method named *IGCN*, which approximates their decision tree principles, and is also based on pre-computing a clustering of all users. We will shortly show the feasibility of a different approach to building a decision tree, which does not require assuming a predefined clustering of the users. Offline experimentation in [17] did not show the adaptive *IGCN* to be more accurate than the static *Entropy0* or *HELF*. However, an online study performed on 468 users did show an accuracy advantage to *IGCN*.

3.2 Decision trees for recommendations

While decision trees are among the most prevalent tools in prediction and classification [8], they did not prove as useful for recommendation tasks. We are aware of an only single case where decision trees were used in a recommender setup [13], though the context and technology there are very different than ours. In [13] a commodity-retail recommendation methodology is explored, where a limited set of target customers is selected based on RFM (recency, frequency, monetary) scores. Then, the well used correlation-based similarities are replaced with NRS (normalized relative spending) values. Finally, a C4.5 decision tree is employed instead of association rules to explore the associations between commodities.

3.3 Cold start of a recommender system

A line of works orthogonal to user rating elicitation involves using external content to overcome the cold start problem of collaborative filtering. These works typically leverage items' attributes and combine them with the CF model in a way that allows treating new items for which no much activity exists on record. Two notable works, which were published recently, are [1, 7].

4. ISSUES WITH CURRENT RATINGS ELICITATION APPROACHES

The various existing rating elicitation approaches (of Subsec. 3.1.1) involve different intelligent selection criteria. Yet, we would like to criticize certain aspects of past approaches:

1. *Arbitrariness of selection criteria.* The criteria presented above come from different, sometimes conflicting, desiderata concerning the seeding items. While popularity, contention and coverage are all sensible criteria, they are still detached from any reasonable end goal of optimizing user experience. Furthermore, to make matters worse, note that none of those desired principles is adequate by itself. This gave rise to arguably less natural combinations of different measures, which essentially lead to a less principled process. Instead, we target a process driven by a formal yardstick, which is well tied with end-user experience, and ideally is both simple and natural. (This criticism does not apply to the optimization approach.)

2. *Independence of selected items.* All above criteria score the utility of each single item independently, and consequently select those of highest utility. No consideration is given to how all these items play in tandem. For example, the second selected item might merely be a version (or, a sequel) of the first - after all, it enjoys the same fundamental properties that got the first item selected. Instead, we desire a system that optimizes the utility of the full set of seed items. (Again, this criticism does not apply to the optimization approach.)

3. *Adaptation to an ongoing dialog with the user.* Static approaches present the same items to all users, without considering the system's growing knowledge of the user being interviewed. Intuitively, we prefer systems that adapt to the earlier answers given by the user, leveraging them to achieve a more effective interview process. Yet, the very few attempts for an adaptive elicitation process could not show a real accuracy improvement.

To resolve these issues, we describe a principled adaptive approach based on decision trees.

5. A DECISION TREE RECOMMENDER

This section describes a fully adaptive bootstrap process, which conducts a dynamic dialog with the user while learning their preferences. This is realized by basing the recommendation system on a decision tree.

5.1 Bootstrapping recommendation by trees

Decision trees are well recognized as useful prediction and classification tools [8]. We show here that they can be as effective for certain recommendation tasks, particularly being a natural tool for bootstrapping a recommender system.

In general, each node of the decision tree evaluates user preference toward a certain aspect, and directs the user along a labeled edge to one of its subtrees based on the user response. Thus, a user follows a path from the root to a leaf that characterizes him. This fits well with bootstrapping a recommender system. The interview process of a new user corresponds to following a path starting at the root by asking the user the questions associated with the tree nodes along the path, and traversing the edges labeled by the user's answers. When this process terminates, the user gets the recommendations modeled by the final node of the path, as will be explained shortly. In our case the decision tree is ternary. Each internal tree node corresponds to a single item on which the user is asked. Then, the user proceeds to one of the three subtrees, according to the user's evaluation of the item. The evaluation is either "like" (by the item's "lovers"), "dislike" (by the item's "haters"), or "unknown." Figure 2 depicts the top three layers of an actual depth-6 decision tree.

As with all CF methods, a decision tree is created and optimized by analyzing past user preferences. When training the tree on a star rating dataset, like the Netflix dataset, we must map the star ratings into the ternary preference space underlying the tree. Thus, we take four and five star ratings for a movie as a "like" indication. The one-, two- and three-star ratings, which fall below the dataset

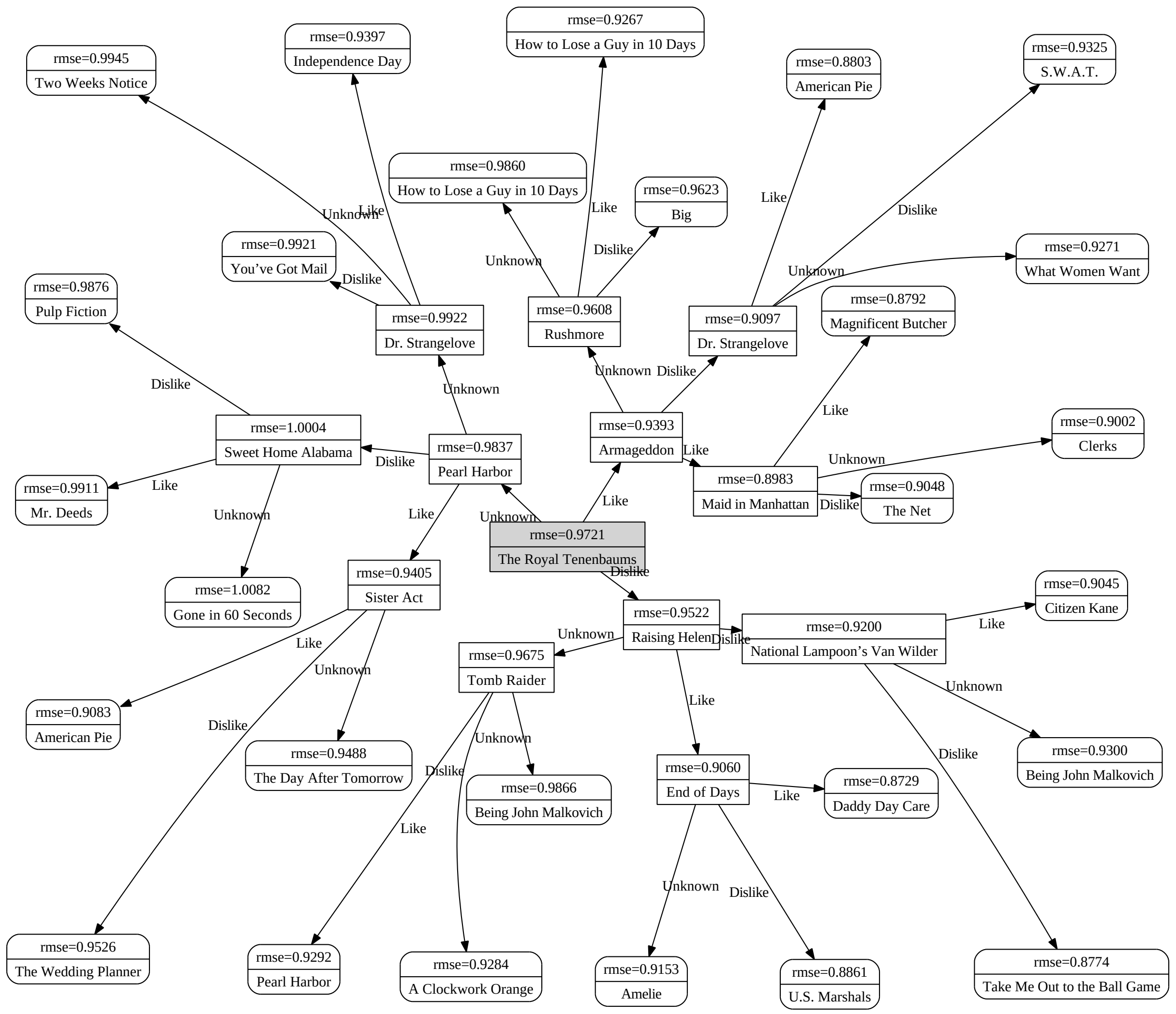


Figure 2: The top three layers of a depth-6 decision tree. A user at the root is asked on the movie "The Royal Tenenbaums" (TRT). On average, by completing up to 6 questions, the tree will predict her rating with RMSE=0.9721. If she likes TRT, she will be next asked on "Armageddon," and the expected RMSE at the completion of the process will decrease to 0.9393. If she did not know TRT, the following question would be on "Pearl Harbor," whereas the expected final RMSE will rise to 0.9837. Those who dislike TRT, will be asked on "Raising Helen" and their expected RMSE would be 0.9522.

average of 3.6 stars, indicate "dislike". Finally, unrated items are taken as "unknown." Other possibilities are discussed in Sec. 8.

Each tree node, whether internal or a leaf, represents a group of users. Consequently, each node of the tree predicts item ratings by taking the mean rating by its corresponding users. The mean prediction minimizes the error associated with each node, which is defined as *the squared deviation* between the predicted ratings at the node and the true ratings for the users represented by the node. Note that here we deviate from common error metrics typical to classification tree construction (like entropy or Gini impurity), but rather rely on the squared loss metric (equivalent to RMSE for optimization purposes), which is common in recommender systems.

Formally, let t be a tree node and $S_t \subseteq \mathcal{U}$ be its associated set of users. The value predicted by the node t to item i is: $\mu(t)_i = |S_t \cap \mathrm{R}(i)|^{-1} \sum_{u \in S_t \cap \mathrm{R}(i)} r_{ui}$. The squared error associated with node t and item i is: $e^2(t)_i = \sum_{u \in S_t \cap \mathrm{R}(i)} (r_{ui} - \mu(t)_i)^2$. The overall squared error at node t is: $e^2(t) = \sum_i e^2(t)_i$.

Construction of the decision tree follows a common top-down practice. For each internal node we choose the best splitting item, namely the item that partitions the users into three sets such that the total squared prediction error is minimized. Accordingly, user population is partitioned among subtrees, and then the process continues recursively with each of the subtrees.

Formally, given a tree node t and a candidate splitting item i, three subtrees are defined $tL(i), tH(i), tU(i)$, corresponding to the partition of the S_t users into three disjoint groups—Lovers, Haters and Unknowns—based on their three possible evaluations of item i. Then, the squared error associated with splitting by i is: $Err_t(i) = e^2(tL) + e^2(tH) + e^2(tU)$. One has to loop over all n items, and choose the splitter as the item minimizing the squared error:

$$splitter(t) \overset{def}{=} \arg\min_i Err_t(i) \qquad (1)$$

Before delving into more technical details, we would like to discuss properties of this process. Since the decision tree is directed by the posterior goal of minimizing RMSE, we do not really need to show that our prior conceptions of an interviewing process hold. Yet, we do find such a discussion insightful to understanding the logic behind the tree construction process.

The ideal squared-loss-minimizing splitter would partition the users into three groups, such that the rating behavior within each group is homogeneous. Earlier we discussed the importance of selecting familiar items for the interviewing process. Indeed, we would expect items selected by (1) to be familiar ones. Otherwise, a non-familiar item will place, by definition, almost all users within the Unknowns user group, effectively transferring the parent node user group almost as a whole into a single child user group. Clearly, this is not the best way to decrease the squared error. We have also discussed the importance of asking the user about controversial items. This is also a property we expect from the splitting items. A non-controversial item will place almost all raters within a single group – either the Lovers group or the Haters group. This cannot lead to the desired refinement that creates more homogeneous user groups. Another desired property is asking about items indicative of others. This is exactly what we would expect from splitting items in the tree. For a single pivot item to create homogeneous user groups, many other items must be correlated with the pivotal one. Finally, the current user group is defined by all previously presented items. The next shown item can significantly refine the existing group only by providing enough new information. Thus, the desired property of presenting a diverse set of items is another by-product of the tree construction process.

5.2 Tree learning algorithm

This section describes the technical details of building the tree. We first explain the general notation and methodology, then move on to describe the required indexing data structures, and finally provide detailed pseudo-code.

A naive construction of the tree would be overly expensive. Consider the selection of a splitting item, by rule (1), which is executed at each node and is the dominant operation in the tree building process. Computation of $Err_t(i)$ involves scanning all ratings by users of S_t, in time $\sum_{u \in S_t} |\mathrm{R}(u)|$. Since each item induces a distinct partition of the users into three groups, the computation should repeat for each of the n items. Thus, choice of the splitting item alone would take time $O\left(n \sum_{u \in S_t} |\mathrm{R}(u)|\right)$ for each tree node. This is too expensive to be practical. For example, in the Netflix data set $n = 17770$, and at the root $\sum_{u \in S_t} |\mathrm{R}(u)| \sim 100M$. Much better performance can be achieved by utilizing the fact that each user only rates a fraction of the total items, about 1% in the Netflix data. This makes the Unknowns users group much larger, on average, than the Lovers and Haters groups. Therefore we save the expensive operations of computing over the large Unknowns groups, instead gathering sufficient statistics from the other groups in order to infer $e^2(tU)$. The computation proceeds as follows.

Let us fix the current tree node t, and the candidate splitting item i for which we calculate $Err_t(i)$. Also, assume that we have already computed the following three types of statistics for node t:

$$\begin{aligned} \forall j \in \mathcal{I}:\ sum(t)_j &= \sum_{u \in S_t \cap \mathrm{R}(j)} r_{uj} \qquad (2) \\ sum^2(t)_j &= \sum_{u \in S_t \cap \mathrm{R}(j)} r_{uj}^2 \\ n(t)_j &= |S_t \cap \mathrm{R}(j)| \end{aligned}$$

The major needed operation is computing these statistics also for the Lovers and Haters groups. That is, computing: $sum(tL)_j$, $sum(tH)_j$, $sum^2(tL)_j$, $sum^2(tH)_j$, $n(tL)_j$, and $n(tH)_j$ for each item j. After this, we can easily derive the statistics for the remaining Unknowns group by subtraction, since:

$$\begin{aligned} sum(tU)_j &= sum(t)_j - sum(tL)_j - sum(tH)_j \qquad (3) \\ sum^2(tU)_j &= sum^2(t)_j - sum^2(tL)_j - sum^2(tH)_j \\ n(tU)_j &= n(t)_j - n(tL)_j - n(tH)_j \end{aligned}$$

These sufficient statistics allow an immediate derivation of the squared loss $Err_t(i) = e^2(tL) + e^2(tH) + e^2(tU)$, since

$$e^2(t) = \sum_j sum^2(t)_j - (sum(t)_j)^2 / n(t)_j \qquad (4)$$

This process is dominated by the computation of the sufficient statistics for the Lovers and Haters. Essentially, for each candidate splitting item i, we scan all ratings of the S_t users who actually rated i. In other words, for each rating a user in S_t provided for a candidate splitting item i, we need to scan all other ratings by this user. This amounts to a running time of $O(\sum_{u \in S_t} |\mathrm{R}(u)|^2)$ for the entire candidate splitter selection process at node t.

Moving on to time complexity of constructing the full tree, observe that at each level of the tree a user belongs to only a single node. Thus, time complexity of computing a single tree level is: $O(\sum_{u \in \mathcal{U}} |\mathrm{R}(u)|^2)$. Since the number of levels in the tree is small in practice (usually 6 or so), this is also the overall time complexity of tree construction. Interestingly, this time complexity exactly equals that of the item-item approach, which also requires $O(\sum_{u \in \mathcal{U}} |\mathrm{R}(u)|^2)$ for computing all item-item similarities [11].

The analysis so far assumed that the dominant operation is selecting splitting items. However, another potentially expensive operation is actually performing the split of users into three sets. Needing to replicate full data structures for each of these three sets would be both time consuming and space inefficient. Hence we opted for an efficient data structure to support the process, using two major indexes. One, called rI, points from each item to its ratings (users and scores), such that $rI[i]$ is the list of all ratings of item i. Similarly, we use a dual index rU, where $rU[u]$ is the list of all ratings by user u. The users-index, $rU[u]$, is global and immutable during the process. As for the item-index, it must support looping over those S_t users who have also rated the current candidate splitting item. To this end, for each item i, tree node t receives all its respective users S_t as a consecutive sequence within $rI[i]$ defined by its left and right endpoints: $left_t(i)$ and $right_t(i)$. Moving from node t to its newly created children tL, tH, and tU, requires an in-place sort of the ratings within $rI[i][left_t(i) \dots right_t(i)]$, for each item i, into separate chunks of Lovers, Haters, Unknowns. This sort takes a linear time as it is based on three discrete values. Hence, time complexity of sorting the relevant portion of the item index is a negligible $O(\sum_{u \in S_t} |\mathrm{R}(u)|)$. Importantly, no copying of the item index is necessary, so the operation is memory efficient.

Based on this indexing, the pseudo code for the tree creation at node t is enclosed within the function GenerateDecisionTree(t).

5.2.1 *Termination*

We would like to elaborate on the termination criteria of GenerateDecisionTree(t). The simplest stopping condition is reaching a predefined tree depth, typically 6 or 7. As we soon show, going beyond such depths yields very limited accuracy gains, thus constituting unwise usage of user efforts. Another case for terminating the recursion is when the best splitting item cannot strictly reduce the squared error: $\min_i Err_t(i) \geqslant e^2(t)$. The last stopping condition

```
**GenerateDecisionTree**($t$)
% Build a decision tree under node $t$ representing user set $S_t$

% Splitting item selection:
for $i \in \mathcal{I}$:
  % Loop over all $S_t$ users that have rated candidate splitter $i$:
  for $u \in rI[i][left_t(i) \dots right_t(i)]$:
    % Determine if $u$ belongs to Lovers or to Haters of $i$:
    $t_u \leftarrow \begin{cases} tL & r_{ui} \geqslant 4 \\ tH & r_{ui} \leqslant 3 \end{cases}$
    % Use user index to loop over all $u$'s ratings:
    for each rating by $u$ — $r_{uj}$:
      accumulate sufficient statistics for $tL$, $tH$ and $j$ per Eq. (2)
  derive all sufficient statistics for $tU$ by Eq. (3)
  calculate $Err_t(i)$ by Eq. (4)
select as splitter item $p = \arg\min_i Err_t(i)$

% Creation of $t$'s subtrees :
if not ready to terminate:
  partition $S_t$ into 3 groups–Lovers, Haters, Unknowns
  % Reorder item-index :
  for $i \in \mathcal{I}$ :
    sort $rI[i][left_t(i) \dots right_t(i)]$ by group attribution
    compute left/right boundaries for subtrees
  create three subtrees: $tL$, $tU$, and $tU$
  recursively call GenerateDecisionTree on $tL$, $tU$, and $tU$
```

is when the number of ratings at the current node drops below a predefined threshold α. That is, $\sum_{u \in S_t} |\mathrm{R}(u)| < \alpha$. As the size of the population shrinks, the expected reliable refinement to be gained is diminishing. Although, thanks to the hierarchical smoothing defined below there is no harm in diving into small populations, the accuracy gain would be minuscule and unjustified. We have found α=200,000 as a good compromise, which saves run time with only a slight effect on accuracy.

The reader would notice that like most learning methods, the behavior of our algorithm is shaped by various meta-parameters, like the tree-depth and α, which have just been mentioned. In order to ensure repeatability, we state the exact settings and values that our system uses in practice. Those values were determined in a greedy manner. That is, when introducing a constant, we executed multiple runs of the algorithms and picked the value yielding the best results. This scheme is not as optimal as a systematic parameter scan, which was not conducted. Hence, the reported values should be taken as a general guideline indicating their desired order of magnitude.

5.2.2 *Accounting for biases*

A significant portion of the observed rating value r_{ui} is attributed to a bias b_{ui}, which is unrelated to the real interaction of user u and item i, but rather to effects influencing u and i in separation from each other. A common practice is assuming an additive parameterization of the bias, where $b_{ui} = b_u + b_i$ [11, 15]. Here, b_u, known as the *user bias*, reflects systematic tendencies for some users to give higher ratings than others. Similarly, the *item bias* b_i captures the inclination of certain items to receive higher ratings than others. The tree learning algorithm, like any CF method, benefits by explaining the biases away, allowing it to deal with the portion of the signal that is truly driven by user preferences. Since our tree construction is driven by minimizing the squared error, the right place to account for biases is when computing that error. This is achieved by subtracting b_{ui} from the rating values. Since squared error, $e^2(t) = \sum_i e^2(t)_i$, is defined by differences between values concerning the same item, any effect of b_i is washed out. Hence, only user biases matter in our computation, and need to be subtracted from ratings.

In practice, biases affect the learning process at only one point - the computation of the $sum(t)_j$ and $sum^2(t)_j$ sufficient statistics. We modify (2) to account for biases as follows:

$$\forall j \in \mathcal{I}: \quad sum(t)_j = \sum_{u \in S_t \cap \mathrm{R}(j)} r_{uj} - b_u \tag{5}$$

$$sum^2(t)_j = \sum_{u \in S_t \cap \mathrm{R}(j)} (r_{uj} - b_u)^2$$

Computation of user biases is done by taking the mean rating of each user, shrunk toward the global mean rating μ:

$$b_u = \frac{\sum_{i \in \mathrm{R}(u)} r_{ui} + \lambda\mu}{|\mathrm{R}(u)| + \lambda}$$

The regularization constant λ prevents over-fitting when a user provides only a few ratings; we use $\lambda = 7$.

A subtle and important observation is that biases are used in order to have a better approximation of the true squared error, while learning the training data. However, they do not influence the format of questions presented to the user, which are still rigid and mapped to fixed star ratings if necessary. This is essential for keeping the interview process simple and particularly important since we do not have any reliable user bias information on users new to the system. Furthermore, note that user bias does not affect the order of ratings for a single user.Thus, while biases are important while constructing the tree in the training phase, they become immaterial during the online phase when recommending top products to the user.

5.2.3 *Random selection*

The function GenerateDecisionTree(t) selects as a splitter the candidate minimizing the squared loss in a fully deterministic manner. However, we have found it useful to randomize the process. First, if users are to try the bootstrapping system multiple times (as we, the authors did!), they would appreciate a different behavior each time. Second, we will discuss in Sec. 7 ways to benefit by combining multiple trees, which are made different because of a randomized learning process.

To facilitate randomization, a node i has a probability proportional to $\max(0, e^2(t) - Err_t(i))^a$ to be selected as the splitter item. This way, items that more significantly reduce the squared loss, have a better chance to be selected as splitters. The constant a dictates how strongly the distribution would be concentrated at those items with the greatest reduction of the squared loss. Picking $a = \infty$ would be equivalent to the original deterministic selection rule. We created 200 randomized trees while setting $a = 10$.

5.2.4 *Parallelization*

Nowadays, commodity multi-core processors are a standard, making it important to consider parallel, multithreaded realization of the algorithm. There are at least two approaches to parallelizing tree construction, which can be combined together.

First, at a more local scope, the process of selecting the best splitting candidate, which dominates the running time, can be executed by multiple parallel threads, each computing the utility of separate candidate items. An advantage of this localized parallelization is implementation ease, without a need to worry about collisions in writing to shared data structures. It also keeps the depth-first-search nature of the tree construction, which is highly memory efficient. Its disadvantage is that it parallelizes only one aspect of the algorithm (though, the dominant one).

A second approach would be allocating a separate thread to each created subtree. An advantage here is that all aspects of the process are parallelized, not only the candidate selection. A disadvantage might be at higher levels of the tree (e.g., the root), where there are not enough subtrees to utilize all potential threads. This is solved by combining the approach with the localized parallelization, which excels at these top levels where any overhead other than candidate selection is minimal. However, a greater issue with allocating separate threads to each subtree is that it changes the nature of the process into the less memory efficient breadth-first-search, and requires duplicate, access-controlled data structures in order to avoid collisions.

Our implementation uses the first, localized parallelization. Recorded benefits in run time are reported in Sec. 6.

5.3 Prediction

The value predicted by node t for item i is denoted by $\hat{r}^t(i)$. It represents a deviation from the (typically unknown) user bias. Consequently all items are ranked by their predicted values. Those users ultimately ending at node t (after following the root-to-t path), will get their recommendations from the top of its ranked list of items. Thus, each node of the tree can either store the full vector of predicted item values, or just the top recommended items. Full details on the prediction and ranking process follow.

5.3.1 Hierarchical smoothing

A well recognized issue with decision trees is that nodes representing smaller populations, usually to be found at deeper layers of the tree, provide inaccurate estimations, which over-fit their limited training observations. A common way to cope with this issue is by pruning the tree, usually involving a post-processing phase [8]. However, pruning a node as a whole would be unappealing in our sparse data scenario, where items have non-uniform rating patterns. Within a single tree node, one item can be supported with a substantial rating population, thereby providing a reliable estimation, while another item is covered by just a tiny population. We thus use a more refined hierarchical smoothing method to avoid over-fitting.

At the root of the tree, q, the predicted values are exactly the averages for the population, that is: $\hat{r}^q(i) = sum(q)_i/n(q)_i$. Then, for each tree node t, the predicted value for item i is shrunk toward the value computed at its parent p:

$$\hat{r}^t(i) = \frac{sum(t)_i + \lambda_1 \hat{r}^p(i)}{n(t)_i + \lambda_1}$$

The shrinkage constant λ_1, which we have empirically set to 200, dictates the pull toward the more robust value computed at the parent (which was already shrunk toward its own parent, unless it is the root). As node t has more users rating i (larger $n(t)_i$), it can make predictions of i with less dependence on its parent value.

5.3.2 Ranking items

Once a user traversing the tree terminates the interview at node t, a plausible recommendation policy would be to recommend to the user the items rated highest at t. However, such a strategy would usually result in presenting very conservative recommendations, as RMSE-based predictors tend to stay near robust averages and typically "play it safe" with generally liked items. This is manifested in our system by the hierarchical smoothing, which is important to RMSE minimization and risk control, by ensuring that big deviations from common values require adequate evidence.

depth	RMSE	time; single thread	time; multi thread
5	0.97260	69:14	12:47
6	0.97172	91:54	16:30
7	0.97129	106:51	18:49
8	0.97115	111:16	19:59

Table 1: Results of a decision tree recommender for various tree depths. Accuracy is measured by RMSE on the Netflix test set. Time is displayed in minutes:seconds, measured on a dual Intel Xeon L5420 2.5GHz machine (total 8 cores).

Our system allows modifying this conservative ranking, in a way that injects more serendipity into it. To achieve this, we identify items that users belonging to the current tree node rate significantly higher than the general population. Formally, denote the rating value of item i attained at the root, q, by $\mu_i \overset{def}{=} \hat{r}^q(i)$. For users at tree node t, define the *differential rating value* for item i as:

$$\hat{d}^t(i) = \hat{r}^t(i) - b\mu_i \qquad (6)$$

Setting the constant b to 0 results in the standard ranking. When $b = 1$, items are ranked by their gap from the overall agreeable mean, so the system suggests those items most unique to the user. Our limited experience (also based on volunteering colleagues in the lab) suggests that higher b values (close to 1) bring better experience. The default setting is $b = 0.8$.

6. EVALUATION

We have run the decision tree learning algorithm on the Netflix training data (100M ratings), and then evaluated its performance on the Netflix test data (2.8M ratings). We list the results in Table 1 in terms of running time and accuracy, for different tree depths. The speedup factor of the simple localized multithreading is about 5.6, on an 8-core machine, driving tree construction time to below 20 minutes. Memory footprint is under 1GB, mostly for holding the two rating indexes discussed above. As can be seen, at least for the Netflix dataset, there is little benefit in going beyond a depth of 6 or 7, as prediction accuracy stabilizes around RMSE=0.971.

To compare the performance of the decision tree learning algorithm to alternative methods for eliciting user ratings, we picked the four methods which have shown best performance in our recent comparative study [5]. These methods include *HELF* and *Entropy0*, which inject popularity awareness into the entropy measure; see [17]. We also included *Var* [5], which ranks items by their variance amplified by multiplying it with the square root of the item popularity: $\sqrt{|\mathrm{R}(i)|} \cdot \mathrm{Var}(i)$. The fourth method is the aforementioned optimization-based *GreedyExtend*.

For each of the four methods, we ordered all movies in the Netflix data, and picked each of the prefixes as a set of movies to be presented to new users. For each resulting seed set, we predict all ratings in the test set, while accounting only for train set ratings on items belonging to the seed set. Error rates of the different criteria are depicted in Fig. 3. In order to contrast performance against decision trees, we ended each plot once reaching the same accuracy level (RMSE=0.971) that a depth 6 decision tree achieves.

Notably, by asking the user on only six items, we get a better accuracy than what methods like Var or HELF could achieve by displaying over 30 items. Similarly, the best performing alternative method, GreedyExtend, requires displaying over 20 items to achieve the same accuracy. This encouraging evidence shows that

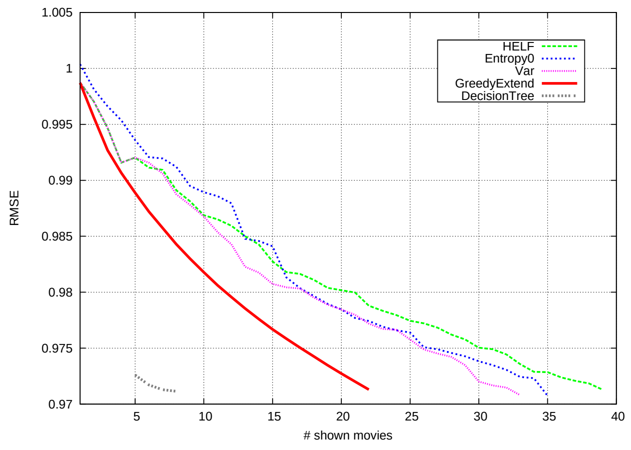


Figure 3: The test error rate vs. number of displayed items (=size of seed set), for various methods of selecting seed set items. It is clear that decision trees require far less item evaluations compared to alternative methods.

a decision tree facilitates an adaptive bootstrapping process with a meaningful advantage over the static alternative.

In another comparison, based on Fig. 1, the first 12 ratings of users were needed in order to match the accuracy of a depth-6 tree. Note that these 12 provided ratings are obviously on items familiar to the user, whereas a decision tree driven interview will most likely include questions on items that are unfamiliar to the user. Thus, by taking the initiative and asking on 6 well selected items, a decision tree provides the same recommendation accuracy that a user enjoys when investing the much greater effort of finding 12 familiar items to rate.

Compared to other (non-bootstrapping) recommendation algorithms, a decision-tree recommender delivers a quite high RMSE. This makes decision-trees undesirable as a traditional recommendation system, and comes at no surprise given that a decision tree allows the user to opine on just the few items she is asked about. In fact, this scenario of asking about a small subset of items, is exactly where the decision-tree excels compared to other recommendation techniques, making it suitable for the bootstrapping task.

We expect that the reported accuracy understates the one experienced by an end user. One can argue that an offline evaluation of a bootstrapping algorithm should be made on users that have a decent amount of ratings on record. This is because many users take a passive standing, and actively provide only a handful of ratings, while actually knowing many more items. However, when elicited by a proactive bootstrapping system, they would provide more answers reflecting their full knowledge. On the other hand, basing evaluation on only heavy raters introduces its own biases, as the number of ratings is likely having a positive correlation with scope of items familiarity. With this disclaimer being stated, we provide the information of tree prediction quality on heavier raters. Indeed, results on such users are far improved, as indicated in Table 2.

7. USING MULTIPLE TREES

There are several reasons to work with multiple trees together. Consider the typical decision tree depicted in Fig. 2. We can see that a user having opinion on the movie at the root can expect a reasonable RMSE of 0.9393 (for "lovers") or 0.9522 (for "haters"). However, if the user did not know the movie at the root, the expected RMSE rises to 0.9837. Consequently, we would like to maximize the chance that a user would know the root item, for at least one tree, by using several different trees together. This way, the user traverses all these trees simultaneously, and for each of the tree layers, we present the user with several items to potentially rate. This increases the chance that the user will be familiar with one of the items displayed at the root, or at a different layer.

#user ratings	RMSE	#test cases
$\geqslant 0$	0.97172	2817131
$\geqslant 10$	0.97003	2773914
$\geqslant 20$	0.96454	2569522
$\geqslant 50$	0.94498	1957842
$\geqslant 100$	0.92175	1414129
$\geqslant 200$	0.90268	903925

Table 2: Partitioning test RMSE by the number of user train ratings. One can argue that users with more train ratings are more representative of the bootstrapping process accuracy, as they have better revealed their scope of knowledge to the system. Results were computed on a depth-6 tree.

Another benefit of using multiple trees is a possible improvement to the user interface. There is evidence that users prefer rating several items together on the same page [16]. This was indeed our implementation choice, where users can choose the number of parallel questions, which would internally equal the number of employed tress; see Fig. 4 for a screen shot.

Looking beyond our specific problem, decision trees are known as "weak learners" and in many scenarios an ensemble of multiple decision trees boosts prediction accuracy. Known variants are Random Forests [3], and Gradient Boosted Decision Trees [4].

Let us assume having K different decision trees (see Subsec. 5.2.3 for details on our construction), while we need to predict r_{ui}. Each of the K trees provides a different estimate of the sought value. We could simply take a prediction as their mean. Even better, we can apply standard blending techniques, which solve a regression problem (by using the test set as a target) assigning a fixed weight to each tree, enabling us to combine them all together according to these weights.

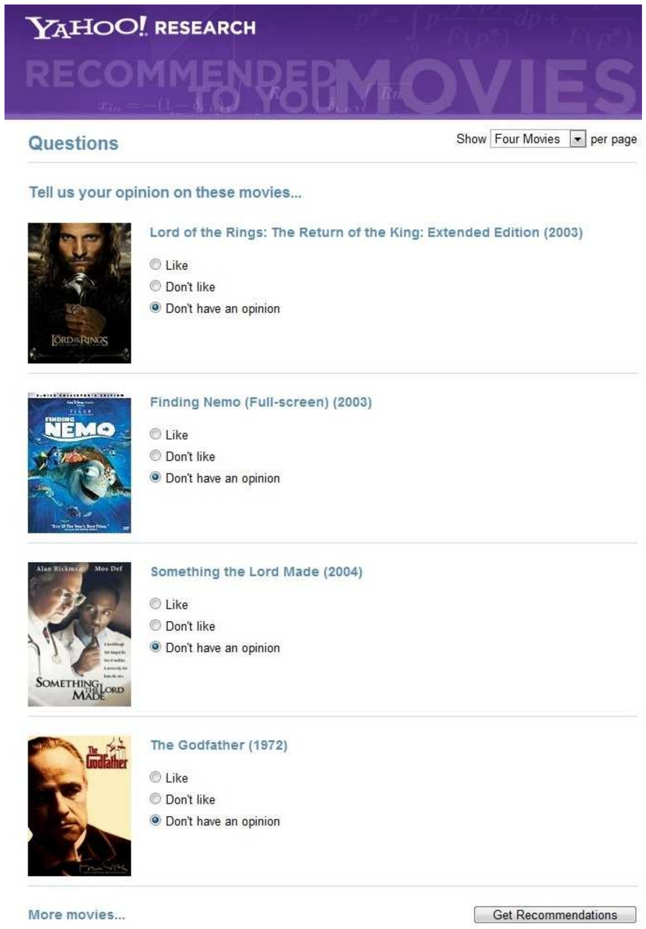
However, all these would miss the unique nature of the trees. Not all tree branches are equal in their contribution. We observe a strong and consistent gap between the value of different user responses. The utility of a "like" feedback far exceeds that of an "unknown" response. Therefore, it is essential to not weight a single tree as a whole, but rather to differentiate between the more and the less useful leaves (or generally, nodes) of the tree.

More specifically, for each of the K trees we need to identify the leaf in which user u falls, which is the one predicting r_{ui}. Thus, instead of dealing with K trees predicting r_{ui}, we now have K leaves predicting r_{ui}: $t_1, \ldots, t_K$. We would like to take the combined prediction as a weighted combination, where more predictive leaves receive a higher weight:

$$\hat{r}_{ui} = \frac{\sum_{l=1}^{K} w_{t_l} \hat{r}^{t_l}(i)}{\sum_{l=1}^{K} w_{t_l}} \qquad (7)$$

Here, w_{t_l} is the weight of leaf t_l, to be defined shortly. Also, recall that $\hat{r}^{t_l}(i)$ is the prediction for item i at leaf t_l.

An essential step is determining the relative weight of each tree leaf. Each leaf (or, node) corresponds to a unique path starting at the root. The predictions at a node tend to be more accurate as there are more "like" edges on this path and less "unknown" edges. Also, nodes at deeper layers (corresponding to longer paths) are doing a more refined job of profiling users, and produce more



Questions

Show Four Movies per page

Tell us your opinion on these movies...

Lord of the Rings: The Return of the King: Extended Edition (2003)

Like [ ] Don't like [ ] [✓ Don't have an opinion]

Finding Nemo (Full-screen) (2003)

Like [ ] Don't like [ ] [✓ Don't have an opinion]

Something the Lord Made (2004)

Like [ ] Don't like [ ] [✓ Don't have an opinion]

The Godfather (1972)

Like [ ] Don't like [ ] [✓ Don't have an opinion]

More movies...

Figure 4: A screen shot of our application: using four trees in parallel, which results in four simultaneous questions to the user. Note that all questions are initialized with a default "don't have an opinion" feedback, which the user may modify.

accurate predictions. We formally quantify this by parameterizing the weight of node t as follows:

$$w_t = (\#L(t) \cdot w_L + \#H(t) \cdot w_H + \#U(t) \cdot w_U)^c \qquad (8)$$

Here, $\#L(t)$ stands for the number of "like" votes in the path leading to t, $\#H(t)$ denotes the number of "dislike" ("hate") votes in this path, and $\#U(t)$ represents the number of "unknown" votes. There are three parameters that need to be computed, which measure the weight of each kind of vote and an exponent of the total sum: w_L, w_H, w_U and c. Values of these parameters were optimized (by an exhaustive search) so as to minimize the blending error. The found values are: $w_L = 5, w_H = 1, w_U = 0.02$ and $c = 2$. Thus, an unknown vote was found to be relatively insignificant. Also, a "like" vote is considerably more useful than a "dislike" vote, which we find to be quite interesting. This may stem from the structure of the dataset, where positive ratings outnumber negative ones and hence are easier to get correlated with others. Importantly, these values were found to be robust to very different values of K (ranging from a few to hundreds).

Once we have placed the mechanism for blending K given trees, the next question is how to choose those trees we want to blend. While we have generated hundreds of trees, selecting them all is not a practical option, as it will overwhelm the user with too many items to evaluate. Typically, we select $1 \leqslant k \leqslant 5$ trees, thereby presenting k questions simultaneously in a single screen. Selection of the k trees involves different considerations. First, we want them to be among the better predicting ones. Second, we need them to complement each other. For example, we would like each one of the trees to present a different question at its root, in order to maximize the chance that the user will be familiar with at least one of the presented items.

All these considerations and others are resolved by posing the problem as a task to best predict the full test set, which uniformly represents all known users. We are looking for a subset of k trees, whose blending according to (7) would minimize the error on the test set. This is performed by forward-selection [8]: first we pick the tree of lowest RMSE, and then successively add the tree which lowers RMSE the most. The way RMSE decreases as a function of k (number of blended trees) is depicted in Fig. 5. Note that with five trees, RMSE decreases from 0.9725 to 0.9656. Going much beyond five trees is not practical for a user-friendly bootstrapping process, but we would still mention achieving a RMSE of 0.9622 with about 30 trees, which later on stabilizes and converges around 0.9617. We also allowed randomizing this process, and preparing different sets of trees, so that repeated users get a different experience each time they use the system.

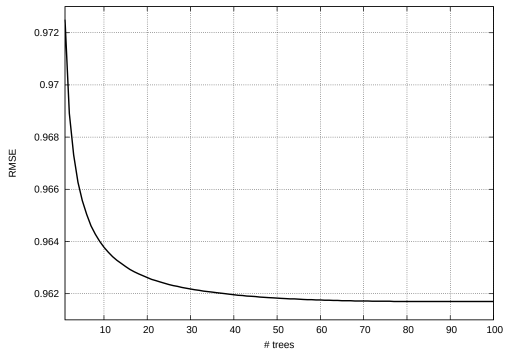


Figure 5: RMSE as a function of the number of blended trees.

8. DISCUSSION AND FUTURE WORK

Decision trees possess several properties that make them fit very well an initial dialog with a user new to a recommender system. By design, they naturally map to an interview process, where all possible response sequences are represented. Furthermore, the greedy tree construction, which optimizes each prefix of a root-originating path, is advantageous. Users may abandon the process at any point, and still be served quality recommendations based on the (possibly internal) node they have reached.

Additionally, decision trees also offer several general advantages that may make them appealing to other recommendation scenarios. First, they do not require numerical ratings, and work well with the more general ordinal ratings (e.g., A-to-F rating scale), as splitting rules only require ordinal values. Trees can also naturally cope with user feedback of various types and granularities, e.g. combining star ratings with a list of liked genres. This is because each internal node is free to apply a different kind of splitting rule. Finally, they represent a hierarchical clustering of users that is easy to store and compute, which may be useful for other applications.

When designing the trees, we opted for 3-way splits corresponding to three possible user responses ("like," "dislike," and "unknown"). Some readers may wonder why we did not work with a more refined split such as a 6-way one matching the common five star levels plus an "unknown" response (see also [17]). There are several reasons to our choice. First, one may argue that in some cases, it is better to give a new, novice user the simplest choice during the sign-up process. A deeper reason is that it is well recognized that decision trees work better with fewer splits. For example, Hastie et al. [8] advocate binary splits, as "Multiway splits fragment the data too quickly, leaving insufficient data at the next level down." Also note that distinguishing between different intensities of affection (4-stars and 5-stars) would likely be futile, as it prevents any linkage between these similar ratings values. Still our choice might be suboptimal. One can reasonably argue for a 4-way split, adding a user response for describing a "neutral" feeling, which would probably map into 3-star rating. We leave this to future work.

In the future we would like to experiment with other cost functions, especially ones related to the quality of the top-K item ranking. The main challenge would be keeping computation efficient, which largely depends on the possibility of a rapid evaluation of the loss function at the large Unknowns user group.

We are also in the process of activating the decision tree process on a larger and differently structured dataset based of music ratings. This dataset includes user feedback not only to basic items, but also to broader attributes such as genres, which are particularly important for a quick bootstrapping process. Once a real life system is deployed, it will facilitate online tests of our process.

9. CONCLUSIONS

Introducing a new user into a recommender system should require a low-effort bootstrapping process, after which the user can immediately appreciate the value provided by the system. System designers trying to impress their new users, who might be the most judgmental ones, should look at methods providing maximal prediction accuracy at minimal distraction to the user. This is usually achieved by conducting a short interview with the user, where she is asked to evaluate certain products or categories.

We have shown a major accuracy improvement when replacing the static bootstrapping process, where a single questionnaire fits all users, with an adaptive one, where the process dynamically changes the interview questions, as ratings are being entered by the user. Adaptive bootstrapping is achieved by a novel decision-tree-based recommender. We detailed certain attributes of the data that accommodate an efficient tree construction process, whose asymptotic time complexity matches that of the classical item-item method. We also discussed various extensions of the basic tree model, which improve its accuracy.

As a concrete example of improvement, our method – after just six user evaluations – achieves a level of accuracy that the best alternative process achieves only after collecting over 20 evaluations.

10. REFERENCES

[1] D. Agarwal and B. Chen. Regression based Latent Factor Models. *Proc. ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 19–28, 2009.

[2] J. Bennett and S. Lanning. The Netflix Prize. *Proc. KDD Cup and Workshop*, 2007.

[3] L. Breiman. Random Forests. *Machine Learning* 45:1, 5–32, 2001.

[4] J. H. Friedman. Stochastic Gradient Boosting. *Computational Statistics & Data Analysis* 38:4, 367–378, 2002.

[5] N. Golbandi, Y. Koren and R. Lempel. On Bootstrapping Recommender Systems. *Proc. International Conference on Information and Knowledge Management (CIKM)*, 2010.

[6] D. Goldberg, D. Nichols, B. M. Oki and D. Terry. Using Collaborative Filtering to Weave an Information Tapestry. *Communications of the ACM* 35:12, 61–70, 1992.

[7] A. Gunawardana and C. Meek. Tied Boltzmann Machines for Cold Start Recommendations. *Proc. third ACM conference on Recommender Systems (RecSys)*, 117–124, 2009.

[8] T. Hastie, R. Tibshirani and J. H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd edition)*. Springer Verlag, 2009.

[9] J. Herlocker, J. Konstan, L. Terveen and J. Riedl. Evaluating Collaborative Filtering Recommender Systems. *ACM Transactions on Information Systems* 22:1, 5–53, 2004.

[10] A. Kohrs and B. Merialdo. Improving Collaborative Filtering for New Users by Smart Object Selection. *Proc. International Conference on Media Features (ICMF)*, 2001.

[11] Y. Koren. Factor in the Neighbors: Scalable and Accurate Collaborative Filtering. *ACM Transactions on Knowledge Discovery from Data* 4:1, 2010.

[12] Y. Koren, R. Bell and C. Volinsky. Matrix Factorization Techniques for Recommender Systems. *IEEE Computer* 42:8, 30–37, 2009.

[13] S. Lee. Commodity Recommendations of Retail Business Based on Decision Tree Induction. *Expert Systems with Applications* 37:5, 3685–3694, 2010.

[14] G. Linden, B. Smith and J. York. Amazon.com Recommendations: Item-to-item Collaborative Filtering. *IEEE Internet Computing* 7:1, 76–80, 2003.

[15] A. Paterek. Improving Regularized Singular Value Decomposition for Collaborative Filtering. *Proc. KDD Cup and Workshop*, 2007.

[16] A. M. Rashid , I. Albert , D. Cosley , S. K. Lam , S. M. Mcnee , J. A. Konstan and J. Riedl. Getting to Know You: Learning New User Preferences in Recommender Systems. Proc. 7th international conference on Intelligent user interfaces (IUI), 127–134, 2002.

[17] A. M. Rashid, G. Karypis and J. Riedl. Learning Preferences of New Users in Recommender Systems: An Information Theoretic Approach. *SIGKDD Explorations* 10:2, 90–100, 2008.

[18] B. Sarwar , G. Karypis , J. Konstan and J. Riedl. Item-based Collaborative Filtering Recommendation Algorithms. *Proc. 10th international conference on WWW*, 285–295, 2001