RecTree:
A Linear Collaborative Filtering Algorithm

by

Sonny Han Seng Chee
M.Sc., University of Toronto, 1992.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTERS OF SCIENCE
in the School
of
Computing Science

APPROVAL

Name:	Sonny Han Seng Chee
Degree:	Master of Science
Title of thesis:	RecTree – A Linear Collaborative Filtering Algorithm

Examining Committee: Professor Qiang Yang
Chair

Professor Jiawei Han,
Senior Supervisor

Professor Wo-shun Luk
Supervisor

Professor Louis Hafer
External Examiner

Date Approved:

Abstract

With the ever-increasing amount of information available for our consumption, the problem of information overload is becoming increasingly acute. Automated techniques such as information retrieval (IR) and information filtering (IF), though useful, have proven to be inadequate. This is clearly evident to the casual user of Internet search engines (IR) and news clipping services (IF); a simple query and profile can result in the retrieval of hundreds of items or the delivery of dozens of news clippings into his mailbox. The user is still left to the tedious and time-consuming task of sorting through the mass of information and evaluating each item for its relevancy and quality. Collaborative filtering (CF) is a complimentary technique to IR/IF that alleviates this problem by automating the sharing of human judgements of relevancy and quality.

Collaborative filtering has recently enjoyed considerable commercial success and is the subject of active research. However, previous works have dealt with improving the accuracy of the algorithms and have largely ignored the problem of scalability. This thesis introduces a new algorithm, *RecTree* that to the best of our knowledge is the first collaborative filtering algorithm that scales linearly with the size of the data set. *RecTree* is compared against the leading nearest-neighbour collaborative filter, *CorrCF* [RIS+94], and found to outperform *CorrCF* in execution time, accuracy and coverage. *RecTree* has good accuracy even when the item-rating density is low – a region of difficulty for all previously published nearest-neighbour collaborative filters and commonly referred to as the *sparsity* problem. Our experimental and performance studies have demonstrated the effectiveness and efficiency of this new algorithm.

Acknowledgments

I would like to thank my senior supervisor, Professor Jiawei Han, for his encouragement and guidance during my studies. His boundless enthusiasm and availability for discussion helped shape this research during the crucial stages. My gratitude also goes to my supervisor Professor Wo-shun Luk and the external examiner, Professor Louis Hafer for providing valuable feedback that has served to improve this thesis.

I would like to thank Compaq Corporation (formerly Digital Corporation) for making the EachMovie database available online. Without this generous gesture, this research would not have been possible.

I would like to thank Geoffrey Bonnycastle Glass for his meticulous proofreading of this work. He guided me to the virtues of the present tense.

Lastly and most importantly, I would like to thank my sweetheart and wife for her enduring support, encouragement, and impetus for the completion of this work.

Dedication

To my Rat, my RatScal and
My wife.

Contents

List of Tables

List of Figures

Chapter 1 Introduction

Collaborative filtering (CF) is a name used to describe a variety of processes involving the recommendation of items based upon the opinions of a neighbourhood of human advisors. **Amazon**[1] and **CDNow**[2] are two well known e-commerce sites that use collaborative filtering to provide recommendations on books, music and movie titles; this service is provided as a means to promote customer retention, loyalty and sales [SKR99]. Despite this tremendous commercial interest, the majority of current research has focussed on improving the accuracy while making only passing reference to the performance and scalability issues. The fastest CF algorithms have quadratic complexity [RIS+94] [SM95] [PHL00] [KM99a] [Paz99]. This thesis introduces, to the best of our knowledge, the first collaborative filtering algorithm, *RecTree*, with linear time complexity. *RecTree* has accuracy and coverage that is superior to the well known correlation-based collaborative filter [RIS+94].

1.1 Information Overload

In our day-to-day activities, we are faced with an overwhelming amount of information. From the moment we wake, we are inundated with requests for our attention and time. How do we choose to spend our limited time on this seemingly endless stream of demands?

We employ several strategies. We rely on spot judgements: quite literally, we judge a book by its cover. Obviously, marketers use graphic effects and advertising to attract our attention and to manipulate our decision. As buyers and consumers, many of us can attest to the

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

[2] http://www.cdnow.com

mixed success of this strategy. We rely on luck: very interesting or valuable items come to our attention serendipitously. Conversely, we can argue that luck saves us from wasting our time on very uninteresting or irrelevant items. As the name implies, this strategy cannot be relied upon. We rely on the opinions of others: we seek advice from those we trust before making a consumption decision. We ask our friends and associates to recommend a good movie to watch. We consult a food critic for a good restaurant to dine at. We rely on the newspaper editor to bring us information that is relevant to us. We rely on the store manager to stock brand items that meet our tastes. This advisory circle helps filter and recommend items for our attention. Advisors who consistently recommend items that we like become more trusted and their recommendations are accorded more weight in our decision-making. Similarly, advisors whose recommendations rarely match our preferences are given smaller weight and eventually leave our circle.

However, our advisory circle is necessarily finite and limited in experience – we can only form a limited number of relationships in our lifetime and each member of the advisory group can sample but a small subspace of all items. Consequently, we often make a decision with inadequate advice. This problem has become increasingly acute with the interconnection of the world through the Internet. Now, we have access to an even wider selection of products and the availability of information sources seems to grow exponentially with no limit. Obviously, our terrestrial strategies for dealing with information overload are inadequate and we need some automated assistance.

1.2 Information Filtering/Information Retrieval

Information retrieval and information filtering (IR/IF) are a group of techniques that have enjoyed widespread success [Sho92] [FD92]. The function of the IR system is described as "leading the user to those documents that satisfy his/her need for information" [Rob81]. IR systems adopt the view that a query is an approximate expression of an information need. Users must engage in an iterative process of modifying and refining their query until the information need is satisfied. By contrast, information filtering adopts the view that user's information interests are stable and can be accurately reflected in a profile [BC92]. Documents that satisfy a profile are extracted from a data stream for presentation to the user.

IF and IR use the same basic process for obtaining document matches. The query and document collection/data stream are initially converted into textual surrogates, typically keywords [BC92]. These surrogates are then matched using one of three major alternatives: Boolean, vector space and probabilistic retrieval models. Vector space methods represent the query and the documents as keyword vectors in a multidimensional space. Terms are weighted by the importance and distribution of keywords in the corpus and query. The query vector is compared against each document vector using a similarity measure, such as the vector cosine [SMc83]. The underlying assumption is that a high similarity of the query and a document vector implies that the document is highly relevant to the informational need. Extensions to the basic vector space method attempt to capture term association and domain semantics [DDF+90] [FD92]. A detailed description of vector space and the other matching methods can be found in [SMc83].

Despite their success, IR/IF have a number of limitations. IF/IR tend to retrieve many items that are irrelevant simply because the keywords are contained in the document. This is readily evident to any casual user of Internet search engines: a simple query string quite often results in hundreds of matches. This problem is somewhat ameliorated by creating more expressive queries, but most users lack either the skill or the patience to pursue this approach. In addition, IF/IR can only be applied to items that are textual or have associated textual attributes [BC92]. It would be impossible to ask an IR system to retrieve music pieces that are "happy sounding". Furthermore, IF/IR cannot readily incorporate subjective judgements into their matches. Measures of quality and style, for example, cannot be represented. Finally, IF/IR systems suffer from "more-of-exactly-the-same" syndrome. Only documents that match on keywords will be presented to the user. A highly relevant document that happens to use different keywords will never by retrieved/extracted.

1.3 Automated Collaborative Filtering

Collaborative filtering (CF) is a new field of study that sprung from the seminal work by [GNO+92] on the Tapestry email system. Collaborative filtering seeks to automate the terrestrial advisory circle that we alluded to in an earlier section. Members of the advisory circle are identified based upon the similarity of their rating history to that of the user. The opinions of the advisors provide recommendations on as-yet unseen items. Unlike the terrestrial advisory circle, CF does not require that a personal relationship exist between the user and his advisors; in most instances, the advisors will be unknown to the user.

Whereas IF/IR seeks similarities between the query/profile and the items, CF seeks similarities between user rating histories and to exploit other users to make recommendations. CF is complementary to IF/IR and does not suffer from some of its limitations. Specifically, CF incorporates subjective judgements into its match. The typical CF query is of the form: “Retrieve movies that I may like”. Secondly, CF does not require a textual representation and can be as readily applied to textual, as well as audio and video content; any item that a human user can evaluate and place a judgement, is amenable to CF. Finally, CF does not suffer from “more-of-exactly-the-same” syndrome. The advisory circle consists of human members with individualistic tastes, which are drawn upon to make recommendations.

Unlike traditional classification and segmentation analysis, CF can be applied where user and item attributes are missing or difficult to obtain. On the Internet, accurate demographic and psychographics information is notoriously difficult to obtain. To some degree this may be due to user concern over privacy and the commercial use of personal information; in a survey of 10,000 households by Forrester Research Inc., two thirds had serious concerns about their privacy [FOR99]. Furthermore, when users submit surveys, over 50% [Gre99] of the applicants provide false information. These issues complicate the analysis of these online forms of data.

In contrast, users seem less reluctant to provide item-rating information. The EachMovie recommendation service [EM97] collected 10 or more movie preference ratings from over 60% of its membership. This same membership however responded poorly when asked their age, gender and zip code. Less than 10% of the membership provided all three pieces of demographic information.

1.4 OLAP and Data Warehousing

OLAP (On-Line Analytical Processing) and data warehousing are two complementary technologies that are enjoying considerable recent commercial success. These technologies rapidly aggregate measures across dozens of dimensions and across all the dimensional permutations. Decision makers can use OLAP to quickly answer questions such as: "What was the number 1 selling baby product, across each province, across each store, in the last 3 years?" This type of query is prohibitively expensive for a relational database to execute. Data warehousing is the process of cleaning data, making it consistent with a data dictionary, and persisting it in a non-volatile data store. While an operational database may be purged periodically (to remove lapsed customers, for example), a data warehouse will never (theoretically) erase any data. New data is appended to it, but old data is never removed. Data warehousing is a data pre-processing step to OLAP.

OLAP and data warehousing is used to implement a popular form of collaborative filtering. Many on-line retail sites customize a web page to include links like: "*X* is the most popular item of type *Y*" or provide statistics such as: "This item has been downloaded *Z* times". OLAP provides the technology to compute these aggregates very rapidly; in some instances the computation can be real-time. Provided that users supply demographic information, other interesting statistics can be computed and displayed to the user. These factoid help the user filter his selection and hopefully, for the vendor, result in a sale. An example of such a factoid is: "89% of professionals in your category in your state buy *thisTypeOf* insurance from *X*". Analyzing the data warehouse for interesting patterns may also be a useful technique for filtering. If a user discovers that a set of items are viewed frequently together, he may decide after viewing the first, that the remaining items in the collection are worthy of consideration; the pattern 'recommended' the set of items to the user. Other data mining techniques such as time series analysis, sequence analysis, clustering and classification can be applied to a data warehouse and the discovered patterns used to recommend items [JCC98].

OLAP and data mining is, however, not inextricably linked to data warehousing. DBMiner is a data mining-OLAP hybrid system that operates directly on relational databases [JCC97]. This system could similarly be used to mine and recommend items to users.

Multimedia-Miner is also a data mining-OLAP hybrid system that operates on multimedia databases [ZHL+98]. This system could be used to discover patterns in multimedia items such as audio or video clips. These patterns could be the basis of recommendations.

1.5 Problem Statement

A CF algorithm makes recommendations to the active user a based on the item ratings of l advisors. Denote the set of all items as M and the rating of user u for item i as $r_{u,i}$ or alternatively $r_u(i)$. Let the vector $\mathbf{r}_u(M)$ denote ratings for all items for user u and the set $Y = \{\mathbf{r}_1(M), \mathbf{r}_2(M), \mathbf{r}_3(M), \dots, \mathbf{r}_n(M)\}$ denote the database of all user item-rating vectors. Define $S_u = \{i | i \in M \cup r_{u,i} = \Theta\}$ as the subset of all items for which user u has not yet rated and consequently for which a collaborative filter may provide predictions. The symbol Θ denotes "no rating". A collaborative filter is then a function f that makes recommendations p_a for the active user a over the set S_a of un-rated items, taking the database of item-rating vectors as input.

$$p_a(S_a) = f(Y, S_a)$$

The function f maps items into real numbers or Θ.

Example 1.

Sammy and her friends are members of a video shop that has established a rudimentary CF system. Each time Sammy and her friends view a video, they can rate the movie on a scale of 1 to 5 indicating their level of enjoyment. A 5 indicates "I enjoyed it very much, I would strongly recommend this movie" while a 1 indicates "I hated it, don't bother". Her rating history and those of her friends are recorded in the table below.

		Titles				
		Starship Trooper (A)	Sleepless in Seattle (R)	MI-2 (A)	Matrix (A)	Titanic (R)
Users	Sammy	3	4	3	3	5
	Beatrice	3	4	3	1	1
	Dylan	3	4	3	3	4
	Mathew	4	2	3	4	5
	Gum-Fat	4	3	4	4	4
	Basil	5	1	5	5	1

Table 1: A compilation of movie ratings for Sammy and her friends. A high score indicates greater preference. The letters R and A following a title denote a romantic and action title, respectively.

Suppose that prior to watching the *Matrix* and *Titanic*, Sammy asked the CF system to choose the movie that best matches her taste. Which movie should the CF system recommend?

A rudimentary CF system could make sensible predictions using the group average. The average rating of *Matrix* is 3 while *Titanic* has an average rating of 14/4. The system would therefore recommend *Titanic* over *Matrix*. We see that this recommendation matches well with Sammy's higher rating for *Titanic* in comparison to *Matrix*.

This rudimentary system falls well short of automating the terrestrial advisory circle. In particular, the group average algorithm implicitly assumes that all advisors are equally trusted and consequently, their recommendations equally weighted. An advisor's past performance is not taken into account when making recommendations. However, we know that in off-line relationships, past performance is extremely relevant when judging reliability of recommendations. Equally problematic is that the group average algorithm will make the same recommendation to all users. Basil, who has very different viewing tastes from Sammy, as evidenced by his preference for action over romantic movies, will nevertheless be recommended *Titanic* over *Matrix*. In the next chapter, we will revisit this example and demonstrate how sophisticated CF algorithms can provide more accurate and personalized recommendations.

□

1.6 Contributions

This thesis describes a new collaborative filtering data structure and algorithm called *RecTree* (an acronym for RECommendation Tree) that to the best of our knowledge, is the first nearest-neighbour collaborative filter that can provide recommendations in linear time. The *RecTree* has the following characteristics:

1. *RecTree* can be constructed in linear time and space.
2. *RecTree* can be queried in constant time.
3. *RecTree* is more accurate than the leading nearest-neighbour collaborative filter, *CorrCF* [RIS+94].
4. *RecTree* has a greater coverage (provides more predictions) than *CorrCF*.
5. *RecTree* does not suffer the *rating sparsity* problem.

We demonstrate the effectiveness and efficiency of *RecTree* through analysis and performance studies.

1.7 Thesis Organization

This thesis consists of five chapters. In chapter 1, we introduce collaborative filtering and the motivation for this work. In chapter 2, we survey related work. Details of the two proposed algorithms, *RandNeighCorr* and *RecTree* are described in chapter 3. In chapter 4, we present experimental results and discuss the strengths and weaknesses of each approach. Finally, in chapter 5, we summarize and discuss future directions for research.

1.8 Nomenclature

Unless it is otherwise specified, this thesis uses the following table of symbols to maintain a consistent discussion:

SYMBOL	DESCRIPTION
$A..Z$	Sets
Y	The set of item-rating vectors.
N	The set of users.
M	The set of items.
l	The number of users; $l \equiv \|N\|$.
m	The number of items; $m \equiv \|M\|$.
n	The size of the data set; $n \equiv lm$.
$\boldsymbol{r}_a$ or $\vec{r}_a$	Rating vector for the active user
$r_{a,i}$	The active user's rating for item i.
$\underline{r}_a$	The active user's average rating. A scalar.
$\boldsymbol{r}_u$ or $\vec{r}_u$	Rating vector for the current user.
$r_{u,i}$	The current user's rating for item i.
$\underline{r}_u$	The current user's average rating. A scalar.
Σ	The standard deviation
Θ	The "no rating" value.

Table 2: Table of Nomenclature.

1.9 Chapter Summary

In this chapter we touched on the problem of information overload and the inadequacy of information retrieval and information filtering techniques for dealing with this problem. Collaborative filtering is a complementary technology that does not suffer from some of the limitations of IR/IF. In particular, CF incorporates human judgements of relevancy and quality by automating the terrestrial advisory circle.

We introduced a database of video ratings that we will serve as a running example in the chapters that follow. A formal statement of the collaborative filtering problem and summary of the contributions of this thesis was presented.

Chapter 2 Related Work

In this chapter we briefly survey the recent developments in collaborative filtering. This is by no means an exhaustive survey.

2.1 Tapestry – the first collaborative filtering system

The phrase “collaborative filtering“ originated from the Tapestry email system [GNO+92]; they define collaborative filtering as a process where “people collaborate to help one another perform filtering by recording their reactions to documents they read.” Tapestry facilitates the sharing of user opinions about documents by allowing them to annotate documents with key phrases. A user receives items by executing a complex query that selects on these annotations and the identity of the annotator. A typical query is: “Show me the documents that Sidney annotated as ‘interesting’”. Tapestry could also use implicit user feedback in its queries. Knowing that Basil sends an email response to only those documents that he finds interesting, we could express a query to select on this action.

Tapestry is essentially a rich querying system where users benefit from the annotations contributed by others – this is the collaborative aspect of the system. The system suffers from a number of limitations, however. Firstly, the user must be aware of the identities or have a prior relationship with his advisors, otherwise he would not think to issue a query based on their annotations. Tapestry provides assistance to make the interaction of terrestrial advisory circles more efficient, but is limited by this very feature. A personal relationship needs to exist between the advisor and advisee. Secondly, users are free to record their reaction using any set of keywords they choose. The unlikelihood of two users using the same set of keywords makes it difficult to create a filter expression that is applicable across the entire user base.

Tapestry is largely a manual system and this is the reason for its limited adoption among the users who participated in the initial study. In the following two sections, we discuss collaborative filters that remove the “relationship requirement” and that limit the user’s manual interaction to providing a numeric rating. In section 2.4 we present some approaches to CF that relieve the user from the explicit task of rating items by inferring endorsements from their interactions.

A collaborative filter is an algorithm that predicts a user’s preference for an item based only on how advisors have rated their preference for it. A collaborative filter differs from other data mining techniques in that the item’s attributes and the users’ demographic information are not necessary for a prediction. This very capability makes CF an attractive technology on the Internet where demographic information is difficult to obtain and anonymity is treasured.

Collaborative filters can be classified into memory-based and model-based algorithms [BHK98]. Memory-based algorithms repeatedly scan the user base to locate other users to serve as advisors. A prediction is then computed by weighting the recommendations of these advisors. An advisor is identified based on his similarity or nearness in tastes to the active user; consequently, these algorithms can be equivalently called nearest-neighbour collaborative filters.

Model-based algorithms infer a user model from the database of rating histories. The user model is then consulted for predictions. Model-based algorithms require more time to train but can provide predictions in shorter time in comparison to nearest-neighbour algorithms. The storage requirements for memory-based algorithms also tend to be somewhat less than those of nearest-neighbour algorithms [BHK98]. In the following sections, we describe some recent collaborative filters.

2.2 Collaborative Filters

2.2.1 The GroupLens Collaborative Filter

The GroupLens system is one of the first automated collaborative filtering systems [RIS+94] to apply a statistical collaborative filter to the problem of Usenet news overload. [RIS+94] assert that user satisfaction with Usenet as a means of disseminating information is declining and that without some assistance to quickly sift the chaff from the wheat, some users are abandoning the medium.

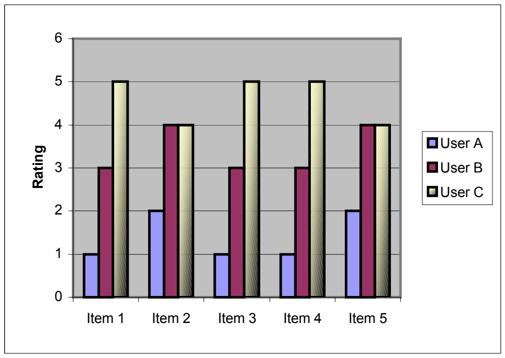
The GroupLens system identifies advisors based on the Pearson correlation of voting history between pairs of users. The Pearson correlation measures the degree with which the rating histories of two users are linearly correlated. Two users may not score items identically, but if they consistently like and dislike the same items then they will have a positive correlation score. Figure 1 shows the rating history for 3 users and their pair-wise similarity coefficients.

An underlying assumption of the GroupLens collaborative filter is that users rate items with a Gaussian distribution; users have an ambivalent preference for most items that they encounter, while for a few items they have a strong like or dislike. The filter takes this behaviour into account by computing the prediction as a deviation from the active user's average rating $\underline{r_a}$:

$$P_{a,i} = \underline{r_a} + \frac{1}{\alpha}\sum (r_{u,i} - \underline{r_u}) \cdot w_{a,u}$$

(1)

The weights $w_{a,u}$ are the pair-wise correlation coefficients between the active user a and the advisor u and the normalization factor α is chosen such that the absolute values of the weights sum to unity.



Pearson Correlation

		User		
		A	B	C
User	A	1	1	-1
	B	-1	-1	-1
	C	-1	-1	1

Figure 1: The rating history for 3 users and their pair–wise correlation.

2.2.2 The Ringo Collaborative Filters

Ringo is a statistical collaborative filtering system [SM95] that provides on-line recommendations for music. Users provide ratings on music albums that they have listened to and the system then recommends a number of music titles. One of Ringo's most popular features is the ability for users to add to the inventory of music titles and artists; this feature was responsible for the nearly 5-fold increase in the item inventory in the first 6 weeks of operation [SM95].

Ringo computes a prediction by taking the weighted average of the advisors' ratings. The weights are computed using one of three different similarity metrics: mean square difference (MSD), Pearson correlation, and constrained Pearson correlation. The MSD metric computes similarity based the mean square difference between rating histories of the users:

$$w_{a,u} = \sum_{N} \frac{(r_{a,i} - r_{u,i})^2}{N}$$

(2)

The Pearson correlation metric has been discussed above and will not be repeated here. The constrained Pearson correlation attempts to take into account the positive and negative endorsements of Ringo's 7-point rating scale (Table 3). Since ratings above 4 are positive endorsements while those below 4 indicate negative endorsements, the correlation metric is modified such that only when both users have rated the item positively or negatively will the correlation coefficient increase. Specifically, the constrained Pearson correlation metric is given by:

$$w_{a,u} = \sum_{Y_{a,u}} \frac{(r_{a,i} - 4)(r_{u,i} - 4)}{\sigma_a \sigma_u \mid Y_{a,u} \mid}$$

(3)

Ringo also uses an alternate prediction algorithm called artist-artist correlation. This algorithm inverts the basic collaborative filtering mechanism by treating albums as potential advisors to other albums. For example, suppose that Basil wants a prediction for the album "Avalon". The artist-artist correlation method computes the correlation between "Avalon" and other albums that Basil has rated and generates a prediction from the weighted average of his scores on those albums.

[SM95] reports that the constrained Pearson correlation metric results in the highest accuracy and highest number of predictions (also known as coverage). The artist-artist correlation metric results in the poorest accuracy and lowest coverage.

RATING	DESCRIPTION
7	BOOM! One of my FAVOURITE few! Can't live without it.
6	Solid. They are up there.
5	Good Stuff.
4	Doesn't turn me on, doesn't bother me.
3	Eh. Not really my thing.
2	Barely tolerable.
1	Pass the earplugs.

Table 3: Ringo's 7-point rating scale.

2.2.3 Personality Diagnosis: A Probabilistic Collaborative Filter

Personality Diagnosis (PD) [PHL00] is based on the assumption that user preferences can be described by a personality type or *true* rating vector, $\boldsymbol{r}_i^{true}$. When users rate items on different occasions, they do so with some deviation about the true value. Gaussian noise is assumed to summarize all of the external factors that affect a rating, such as the user's mood and the context of any other titles rated in the same session. Specifically, the probability that user *a* assigns a rating score of *x* on an item *i* is given by the normal distribution with mean *y*:

$$\Pr(r_a(i) = x \mid r_a^{true}(i) = y) \propto e^{-(x-y)^2/2\sigma^2}$$

(4)

The probability that the active user's true preferences are those represented by another user's ratings is used as measure of similarity and can be computed by applying Bayes' rule. Specifically, the probability that the active user *a* is of the same personality type as another user *i*, is given by the product of probabilities that the active user's rating on each item is normally distributed about the *true* values as given by the user *i*:

$$\Pr\left(\vec{r}_a^{\,true}(M) = \vec{r}_i(M) \mid r_a(1) = x_1, ..r_a(m) = x_m\right) \propto \Pr\left(\vec{r}_a^{\,true}(M) = r_i(M)\right) \prod_{j=1}^{m} \Pr\left(r_a(j) = x_j \mid \vec{r}_a^{\,true}(j) = r_i(j)\right)$$

(5)

The first term on the right hand side of (5) is the prior probability that the active user votes according to vector ***r****ᵢ(M)* and is assumed to be a random variable with a value of $^1/_l$ where *l* is the number of users. Given these similarity scores, a rating probability distribution can be computed for any item and taking the most probable rating in the distribution then generates a prediction. The probability for each rating is simply the sum of probabilities of all personality types that support that rating score:

$$\Pr(r_a(i) = x_i) = \sum_{Y_u} \Pr(\vec{r}_a(M) = r_u(M)) \qquad Y_u = \{u \mid u \in N \text{ and } r_u(i) = x_i\}$$

(6)

PD can equivalently be interpreted as reconstructing the active user's true preferences by taking one of the other users at random and adding Gaussian noise to it. Given the user's rating history the probability that he is actually one of the other users is inferred. [PHL00] reports the accuracy of PD is competitive with those of other nearest-neighbour collaborative filters [BHK98], however the two performance studies were conducted on separate data sets.

2.2.4 The Cluster Collaborative Filter

Recently a collaborative filter based on the weighting of clusters was proposed [KM99a]. This approach applies a hierarchical divisive clustering algorithm to partition the user base into successively finer partitions until a cluster distortion threshold is satisfied. Cluster distortion is defined as the sum of the distance of all data points from the centre of the cluster. Locating the leaf node where a user resides and then taking a weighted average of each cluster's recommendation on the path from the leaf partition to the root generate a prediction. A cluster recommends an item based on the average rating of all its members. Cluster distortion is used to weight each cluster's recommendation. [KM99a] reported that the overall performance was competitive with the correlation-based collaborative filter [RIS+94].

2.2.5 Collaborative Filtering using Clusters

A straightforward application of clustering techniques to ratings data was recently reported [UF98]. The authors applied the KMeans and Gibbs Sampling clustering algorithms to create user and item clusters. Users of the same cluster acted as recommenders for each other. The user clusters were trained by clustering on items purchased by users and item clusters were trained on the users they were purchased by. A scheme of repeated clustering was reported where a 2nd and

3rd iteration of user and item clustering was performed on the item and user clusters created from the previous iteration. The authors argue that repeated clustering on clusters, as opposed to clustering on the actual users/items, has the potential for creating useful neighbourhoods from which recommendations can be drawn. Initial results for repeat clustering are, however, inconclusive. Application of the clustering algorithm to the CDNow music online retail site reportedly doubled email respondents to a music promotional.

2.2.6 Bayesian Classifier for Collaborative Filtering

The naïve Bayesian classifier computes the probability of membership in an unobserved class *c* based on the "naïve" assumption that ratings are conditionally independent [BHK98]. The underlying assumption is that there are certain classes or types of users that have a common set of preferences and tastes. The probability model relating the joint probability of class membership and ratings is given below:

$$\Pr\left(C = c, r_a(1),..,r_a(m)\right) = \Pr(C = c)\prod_{i=1}^{m}\Pr\left(r_a(i) \mid C = c\right)$$

(7)

The left hand side of expression (7) is the probability that a user *a* with the rating history $r_a(1),..,r_a(m)$ is of class *c*. The probability of class membership $\Pr(C=c)$ and the conditional probabilities $\Pr(r_a(i)|C=c)$ are estimated from the training set of user item-ratings. However, the class labels are not directly observed and methods that can infer model parameters with hidden variables must be employed. The EM algorithm was selected to learn the model structure with a fixed number of classes.

The Bayesian classifier was found to be competitive with the correlation-based classifier in accuracy and coverage. However, the classifier has a significantly longer training time.

2.3 Open problems in CF

Despite the growing commercial interest in collaborative filtering, there still remain a number of open problems that have yet to be adequately resolved.

2.3.1 The Early-Rater Problem

A collaborative filter does not provide any benefit to a user if she is the first person in her neighbourhood to rate an item. [AZ97] has speculated that even if the cost of rating an item were zero, most users will still prefer to benefit from others ratings rather than supply ratings themselves. Without a compensation mechanism, CF systems depend upon the altruism of their members to overcome the early rater problem.

2.3.2 The Sparsity Problem

Collaborative filtering systems require a "critical mass" of users to join and provide ratings before they can provide predictions of reasonable accuracy and coverage. Even when there is a large membership, a sufficient number of users must rate each item. The accuracy of many collaborative filters fall below that of non-personalized recommendation via population averages when the rating density is low [RIS+94] [KM99a] [GSK+99] .

2.3.3 The Scalability Problem

The recent collaborative filter research has focused on improving accuracy and largely ignored the problem of execution time. We believe this may be due to two factors. Firstly, the focus of early collaborative filter research was to prove their merit through the accurate prediction of user preferences. As long the execution time was not prohibitive, accuracy was the driving factor in CF research. Secondly, the memberships in the early CF systems were relatively small and consequently satisfactory responses could be achieved by increasing the number of computer resources devoted to the task. However, given the quadratic time complexity of the fastest collaborative filters and the inherent less-than-linear gain in throughput for an incremental increase in computer resources, we believe this is not a problem that can continue to be ignored.

2.3.4 Combining Filter Bots and Personal Agents with Collaborative Filtering

Two recent promising approaches to the *early rater* and *sparsity* problem attempt to increase the rating density by using bots and agents. These automatons submit ratings to the collaborative filtering system as if they are legitimate users. They differ from real users in that they rate every item in the inventory but never ask for recommendations. The relevancy of their recommendation to a human user is automatically determined by the similarity weight that the CF system computes.

[SKB+98] reports the use of "filter bots" in a Usenet application where news items are assigned ratings based upon simple rules. The bots assign a rating based on the proportion of spelling errors, the length of the messages, and the length of included messages. The rationale for these bots is that Usenet members prefer messages that have few spelling errors, are brief and contain more new content in comparison to included messages. Each of these bots rank the entire set of messages and then partitions them into 5 rating bands. All items within a band are assigned the same score. The proportion of news items in each band is chosen to match the human rating distribution of that newsgroup. The spell-checking bot is reported to improve the accuracy and coverage in 4 of 5 newsgroups, while the remaining bots have mixed results. The authors postulate that users do not necessarily care about error free messages, but that good spelling correlates well with such valuable attributes as careful writing style or simple vocabulary.

[GSK+99] reports the use of agents and bots in a movie recommendation application. A number of simple genre bots are constructed that rate a movie with the maximum preference score of 5 if it is of a particular genre or the minimum preference score of 1 otherwise. For each user, three information filter agents are trained on only the movie cast names, the movie's keywords, or the cast names and keywords, respectively. The agent profiles are constructed by computing the TF-IDF (term frequency-inverse document frequency) [SMc83] vector from the collection of movies that the user has rated. Each movie is also converted into a TF-IDF vector. The agents then score each movie by computing the vector cosine distance between their profile and the movie's TF-IDF vector. The movies are ranked and divided into 5 rating bands in proportion to the human rating distribution; all movies within a band are assigned the same score. The results of this study are promising and show substantial improvements in accuracy and coverage when all the bots and IF agents participate in the collaborative filtering system with the 50 users in the study group.

[GSK+99] noted that despite the promising results of the study, the scalability of the proposed solution had not been addressed. The MovieLens and other collaborative filtering systems could not cope with agents and bots that provide such a volume of recommendations nor agents that changed their ratings periodically to reflect their learning of the user's preferences. Furthermore, these approaches are based on information filtering techniques and consequently are constrained by IR's limitations.

2.3.5 Fab

Fab is a web-page recommendation system that solves rating sparsity by computing user similarity from profiles rather than item-ratings [BS97]. A user profile consists of a TF-IDF vector built up from the documents that the user has rated. The vector cosine metric between user profiles, which ranges between –1 and 1, is computed to identify advisors; a value of 1 indicates a perfect similarity, while –1 indicates complete dis-similarity (a user whose tastes are completely opposite to the active user), and 0 indicates a complete lack of interest overlap. Web pages that are highly rated by an advisor are provided to the active user as recommendations. This approach ameliorates the sparsity problem since content similarity is used as a basis of computing user similarity.

Fab's architecture, shown in Figure 2, consists of two families of agents. Collection agents gather documents from the web and deposit them into a central pool. Selection agents match documents to their profiles and deliver them to their users. When a user rates a document, both the selection and the collection agents receive the relevancy feedback. Collection agents receive feedback from all users while a selection agent receives feedback from only its "owner". This feedback strategy results in collection agents that are trained to serve the needs of a group of users while the selection agents are trained to serve the needs of a single user. Collection agents that consistently receive low feedback scores are periodically eliminated and agents with high feedback scores are duplicated. Fab's collaborative aspect arises when a user rates a document highly. At this point the document is forwarded to all of his advisees' selection agents as recommendations. The selection agents make the final decision as to whether to forward or filter a recommended item.

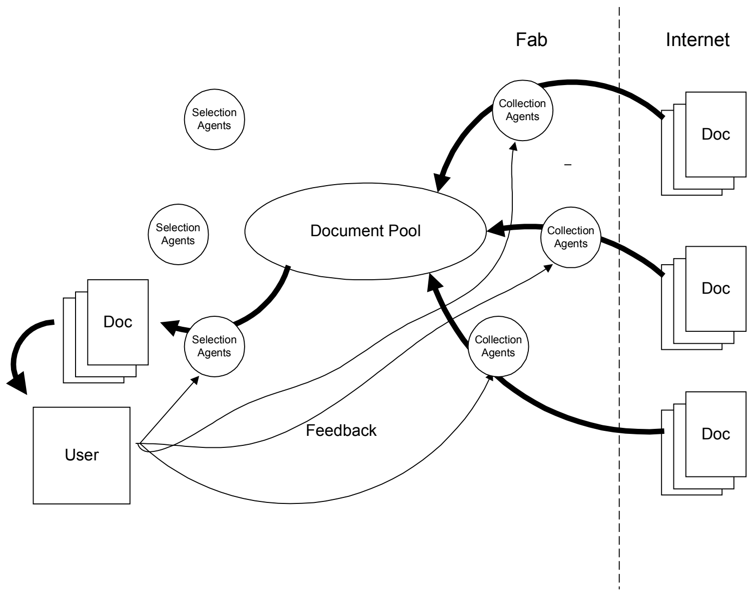


Figure 2: Two families of agents filter documents. Users collaborate by passing on highly rated documents to their neighbour's selection agents.

2.4 CF Applications

2.4.1 SiteSeer

SiteSeer is a recommendation system that predicts preferences for web pages [PR97]. Web-page ratings are implicitly gathered by examining a user's bookmark and bookmark folders. A bookmark is taken as a positive endorsement for a page and a folder provides a context for computing interest similarity and for providing recommendations.

SiteSeer assumes that each folder represents a distinct interest and that a high degree of overlap between users' folder contents are indicative of common interest; a user can therefore belong to as many interest neighbourhoods as bookmark folders. Recommendations are generated by taking the most often saved bookmark in the neighbourhood of folders with a high degree of overlap. SiteSeer treats the bookmarks as purely unique identifiers and does not derive any semantic value from the titles of folders, the bookmarks nor the contents of the web pages. The bookmark folders, however, are used to contextualize the recommendations. In Figure 3, a neighbourhood is formed around Basil's "vacation spots" folder with Sammy, Bea and Dylan. SiteSeer recommends to Basil the link http://www.belize.org within the context of the "vacation spots" folder since it appears in all three of his neighbour's folders.

Using bookmarks as inputs to the collaborative filter has a number of advantages. Bookmarks are less "noisy" data in comparison to mouse-clicks since they represent a conscious investment in time to create and organize. Furthermore, bookmarks are contextualized by the organization of folders providing a natural means for recommendations.

Bookmarks also have a number of limitations. Valued sites are not always bookmarked if they can be reached through other methods such as a search engine, an index page, or if it has a simple URL that can be easily remembered. In addition, bookmarks are created for a number of different reasons, ranging from long-term interest to a short-term need to return to a page. Finally, bookmarks register positive endorsement for a page, but there is no means for registering partial or negative endorsement.

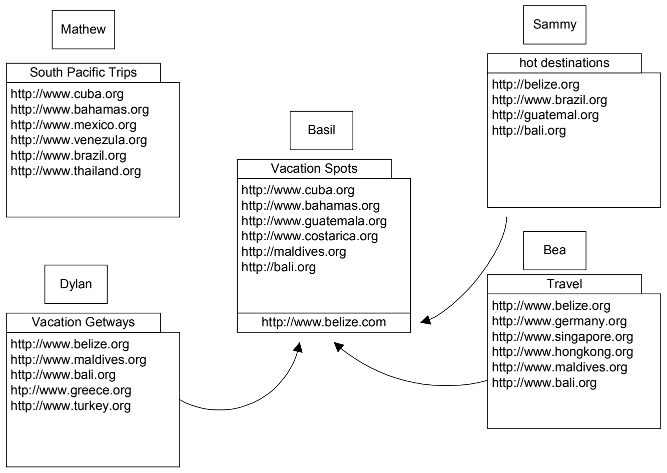


Figure 3: SiteSeer identifies an interest neighbourhood around folders that have a high degree of overlap in bookmarks.

2.4.2 ReferralWeb

ReferralWeb is a system for recommending people (experts) and documents based on the social network [KSS97]. Unlike other automated collaborative filtering systems where the recommenders are anonymous, the identities of recommenders in ReferralWeb are explicit. This is crucial for a determination of trustworthiness and quality of the information source.

ReferralWeb exposes the referral chain as it serves two important functions: it provides a reason for the expert to respond to the requester by making the relationship explicit (perhaps they have a mutual friend or mutual collaborator), and it provides a means by which the searcher can evaluate the trustworthiness or quality of the expert. ReferralWeb does not require the user to explicitly enumerate his contacts, but builds its social network by mining documents published on the Internet.

When a user registers with the system, ReferralWeb constructs a social network by retrieving all documents mentioning the user and then extracting the names of other individuals. This process is recursively applied several times and the results merged into a network. The network can then be used to constrain and filter the search for people and documents. The social network can be used to infer the quality of documents; the underlying assumption is that authors of high quality documents tend to be collaborators or colleagues. For example: "Retrieve database documents by colleagues of J. Gray." Similarly, a query to locate an expert in the data mining field could be expressed as: “Show me the names of individuals who have published a document about data mining that are colleagues or colleagues of colleagues of J. Han.”

ReferralWeb’s obvious disadvantage is that it can build a social network only around individuals who have published online or are mentioned in online press releases, for example academics, CEOs, or spokespersons. However, there is a growing trend for individuals to create an online presence and this will help to extend ReferralWeb’s applicability.

2.4.3 PHOAKS

PHOAKS is an on-line recommendation service that mines Usenet postings as a source of web-resource endorsement [THA+97]. Using a complex library of heuristics, the authors report an 88% accuracy in extracting web-page endorsements from Usenet postings. The basic strategy consists of three steps. Firstly, if the URL contains the word markers that indicate it is being recommended as opposed to being advertised or announced, it is counted as an endorsement. Secondly, if the endorsement has been cross-posted to many newsgroups then it is discarded – such a message is assumed to be too general to be thematically close to any of the groups. Finally, the web-page endorsement is discarded if it is part of the poster’s signature file, embedded in a message or endorsed by the same user in the past; these strategies deprecate self-promotions and double counting.

PHOAK predicts the quality of a web-resource based on the number of distinct users that recommend it. The authors test this assumption of quality by comparing the recommended resources with those in newsgroups FAQs; domain experts maintain FAQs and typically direct users to high quality web pages. [THA+97] report a positive correlation between the number of distinct recommenders to the probability that it will be mentioned in a FAQ.

PHOAKS is still operational and can be accessed on-line at www.phoaks.com

2.5 Chapter Summary

In this chapter we have summarized the recent research in collaborative filtering. We described a number of nearest-neighbour and model-based collaborative filters and indicated some of their strengths and weaknesses. We presented the open-problems in CF and some CF applications.

Chapter 3 RecTree

In this chapter, we discuss our approach to designing collaborative filters with faster execution times. We begin in Section 3.1 with a discussion of preliminary observations that motivate our approach and the assumptions that we make. The following three sections present successive refinements to the basic approach. Section 3.4 presents *RecTree*, a new data structure and algorithm that can deliver recommendations in linear time.

In this and subsequent chapters, the term "item-rating vector" is used interchangeably with user since each item-rating vector is uniquely identified with a user. Therefore, when we say the user $\boldsymbol{r}_u$, it is understood that we mean the user u associated with that item-rating vector.

3.1 Observations and Assumptions

3.1.1 Partitioned Collaborative Filtering

The fastest collaborative filters to date have quadratic complexity with data set size [RIS+94] [SM95] [PHL00]. Each of these algorithms differ in the mechanism of computing a prediction and the metric used to gauge similarity. However, they all require an exhaustive search over the database of l users to locate advisors. To generate predictions for a test set of q users requires lq passes over the database. Obviously, a CF system is required to make predictions for all subscribing users, otherwise excluded users would derive no utility from the system and hence cease to be members. The test set therefore consists of the same users as the database of candidate advisors - yielding the quadratic complexity.

Clearly, we can only hope to improve the complexity of any proposed algorithm if we can avoid or limit the exhaustive search for advisors. We take the latter approach and describe three algorithms below that limit the neighbourhood over which advisors are sought. Each of these algorithms attempts to create partitions of independent neighbourhoods of approximately similar users. Predictions are computed from only the recommendations of members of the same partition. Our general approach can thus be described in the following framework:

1. Phase I: Partition the data set.
2. Phase II: Train the collaborative filter.
3. Phase III: Generate predictions.

Phase II of the framework makes no assumption about the type of collaborative filter used. Nearest-neighbour and model-based algorithms can be inserted without difficulty. A recent empirical study [BHK98] into several nearest-neighbour and model-based collaborative filters reports that both types of algorithms have comparable accuracy. Differences between the algorithms arise in terms of execution time and space cost; nearest-neighbour filters execute significantly faster but require somewhat more disk space. We rationalize that disk space, unlike time, can always be purchased and given the continuing improvements in storage technology, at a steadily declining per unit cost. This thesis focuses on improving the execution time for nearest-neighbour collaborative filters. Phase II can alternatively be paraphrased as: "Train nearest-neighbourhood CF."

3.1.2 In-Memory Algorithms

For this work, we assume that all of the rating data will fit into memory. This is not an unreasonable assumption, as we argue here.

The EachMovie recommendation service[3] accumulated almost 3 million ratings over an eighteen-month period. The service used a 6-point rating scale that is typical of recommendations systems[4]. The total cost to store 3 million ratings is therefore only 3 mega-bytes. If we assume an operating system with the capability to address 1 gigabyte of memory space, an in-memory algorithm should be able to accommodate a service 300 times larger than the EachMovie service. We therefore believe that in-memory collaborative filters can adequately serve the short and intermediate term needs of a recommendation service. It should be pointed

[3] The EachMovie recommendation service was modestly sized with a membership of 73,000 users and an inventory of 1600 items.

[4] There is evidence to suggest that reliability of the data collected does not substantially increase if the number of choices exceeds seven [RR91].

out that our 1 gigabyte memory limitation is somewhat conservative - the next generation of 64-bit operating systems will remove this limitation altogether. An extension of this work to handle very large databases is nevertheless discussed in Chapter 5.

3.1.3 Overview of *RandNeighCorr*, *KMeansCorr* and *RecTree*

Starting in section 3.2 we describe three algorithms, *RandNeighCorr*, *KMeansCorr* and *RecTree*, with the potential for delivering recommendations in linear time. *RandNeighCorr* is the most naïve approach to partitioned collaborative filtering. We present it as the baseline algorithm against which the other two partitioned filters can be compared. *RandNeighCorr* partitions the data set by randomly assigning users to partitions in phase I. In phase II, it computes the similarity between users using the Pearson correlation measure. Finally, in phase III it generates predictions by taking a weighted deviation from the user's mean rating.

KMeansCorr represents a refinement on *RandNeighCorr*'s simplistic partitioning strategy. It replaces the partitioning stage (phase I) with the *KMeans*$^+$ clustering algorithm. Phase II and Phase III of *KMeansCorr* are identical to those of *RandNeighCorr*. A complexity analysis of *KMeansCorr* will reveal that despite being able to execute in isolation the partitioning and training phases in linear time, the combined action of all phases results in at best quadratic complexity.

The *RecTree* algorithm partitions the data in phase I by recursively calling *KMeans*$^+$ to split the data set into child clusters. The chain of intermediate clusters leading from the initial data set to the final partitioning is retained in the *RecTree* data structure, which resembles an unbalanced binary tree. The collaborative filter is trained in Phase II by computing the pair-wise similarity coefficient within each leaf partition using a more accurate similarity metric that we call *correlation*$^+$ for clarity. The *RecTree* generates a prediction by employing a dual strategy. If the user is located in a partition with a *sufficient* number of neighbours, taking the weighted deviation from the mean generates a prediction. If the user is located in a small neighbourhood, *RecTree* generates a prediction by taking the neighbourhood's average rating.

3.1.4 Example Data Set

Throughout this chapter we will return to the fictitious video store, first introduced in Chapter 1, as a running example. The values in the shaded cells are withheld as testing data and the remainder of the table is used for training the proposed collaborative filters. We will on occasion refer to the table below, without the values in the shaded cells, as the video database training set.

Similarly we will refer to the shaded cells as comprising the video database test set. The prediction task will be to predict Sammy's and Basil's ratings for the movies *Matrix* and *Titanic*.

		Titles				
		Starship Trooper (A)	Sleepless in Seattle (R)	MI-2 (A)	Matrix (A)	Titanic (R)
Users	Sammy	3	4	3	3	4
	Beatrice	3	4	3	1	1
	Dylan	3	4	3	3	4
	Mathew	4	2	3	4	5
	Gum-Fat	4	3	4	4	4
	Basil	5	1	5	4	5

Table 4:The fictitious video store ratings database. The shaded cells are withheld for testing, while the remainder of the table is submitted for training the collaborative filter.

3.1.5 Over-partitioning

One may imagine that by partitioning the data into smaller subsets we may be able to obtain any desired performance. However, as Figure 4 shows, an *over-partitioning* of the data adversely affects the coverage (the number of predictions that can be generated); a small neighbourhood means that there are fewer advisors from which to draw recommendations.

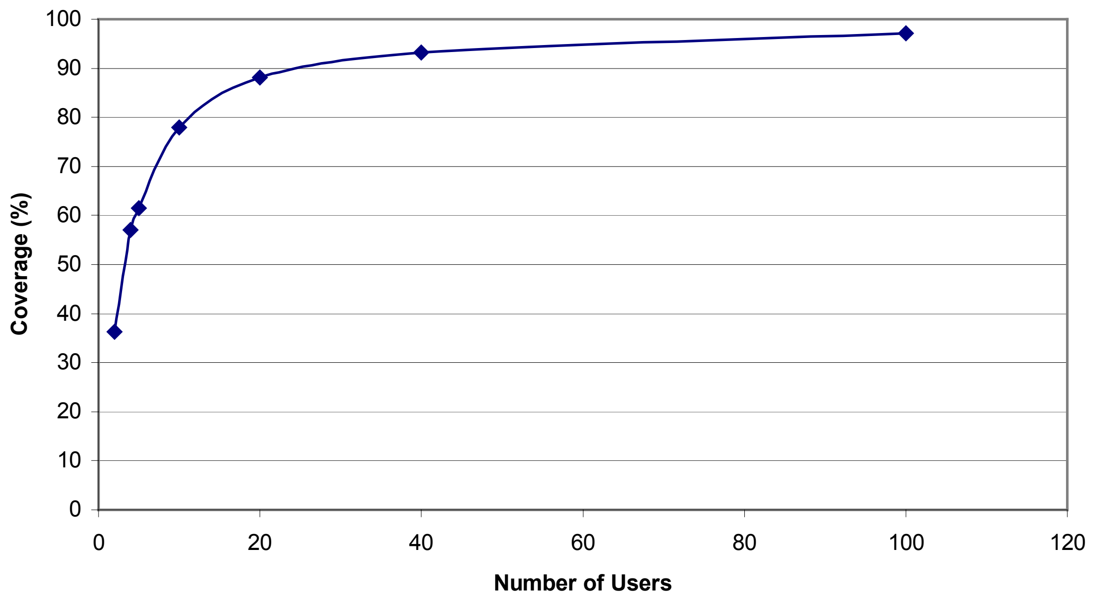


Figure 4: The coverage for all nearest-neighbour collaborative filters declines with smaller training set.

3.2 *RandNeighCorr*

3.2.1 Overview

In this section, we present the most naïve approach to partitioned collaborative filtering to serve as the baseline. This algorithm partitions the database by randomly assigning users to *k* independent partitions. Within each partition we train a correlation-based collaborative filter by computing the pair-wise similarity coefficients between members. Then, we generate a prediction by taking the weighted deviations from the rating mean.

RandNeighCorr will serve as our baseline algorithm against which the other proposed partitioned algorithms must surpass to be worthy of further consideration. In the following subsections we describe each phase of *RandNeighCorr*.

Algorithm 1. ***RandNeighCorr(Y,S,k)***

Input: Y is the training set and is the database of all user item-rating vectors. S is the test set and is the database of all user "no-rating" item vectors; each vector is the set of items for which the user has yet to rate and for which we would like *RandNeighCorr* to produce predictions. k is the number of partitions to create and $k << |Y|$.

Output: A mapping of each element of each vector of S onto a rating score or "no rating", Θ.

Method: The *RandNeighCorr* algorithm is implemented as follows.

```
Phase I: Partition the data set.
        1. Randomly assign each member of $Y$ to one of $k$ partitions.
Phase II: Train the collaborative filter
        1. For each partition $Y'$, call ComputeCorrelationSimilarity(Y').
Phase III: Generate predictions.
        1. Call ComputeDeviationFromMeanPrediction(S)
```

3.2.2 Partitioning Phase

[KMM+97] reports that anecdotal evidence supports a strategy of randomly partitioning users into neighbourhoods and then applying a collaborative filter within each partition. They claimed that this strategy leads to an improvement in speed over the un-partitioned case and yields "useful predictions". However, they presented no performance results. We choose random assignment as it represents the most simplistic approach to partitioning. Our subsequent approaches to partitioning in later algorithms must result in neighbourhoods of more correlated users than random assignment, otherwise the clustering algorithm's contribution must be suspect.

Example 2.

We take our fictitious video database and randomly assign users to one of two partitions.

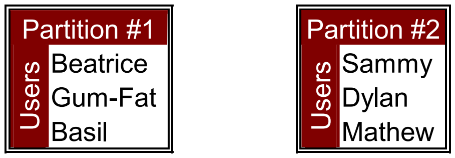


Figure 5: A random bi-partition of the video user base.

□

Corollary 1. *RandNeighCorr*'s partitioning phase has O(*l*) complexity.

Proof. A database of *l* users is randomly assigned to *k* partitions in a single scan of the database.

□

3.2.3 Training Phase

In the training phase we can select from a number of similarity metrics to train our collaborative filter. The recently published metrics include: the mean square difference (MSD), constrained Pearson correlation [SM95], Pearson correlation [RIS+94], vector cosine [BHK98], and personality diagnosis [PHL00]. [BHK98] reports that among the first three of these metrics, the Pearson correlation consistently gave the most accurate predictions. [PHL00] subsequently reports better prediction accuracy with personality diagnosis (PD), but his performance studies were inconclusive as he compares the performance of PD against the other similarity metrics using different data sets.

We choose the Pearson correlation metric as it measures the linear relationship between the rating behaviour of the active user and his advisor. A linear metric is desirable since in the prediction phase we will specify a linear function to map advisors' ratings to a recommendation. The subroutine *ComputeCorrelationSimilarity()* computes the pair-wise correlation between members of a partition Y. The user's mean rating $\underline{r}_u$ and standard deviation σ_u are computed over the set of items that both users have rated in common.

Subroutine 1: ***ComputeCorrelationSimilarity(Y)***

Input: Y is a set of user item-rating vectors.

Output: A square matrix of pair-wise similarity weights $w_{i,j}$ between all members.

Method: The similarity weights are computed in the following procedure.

For each user $\boldsymbol{r}_{a,}$ compute the pair-wise similarity $w_{a,u}$ to each other member $\boldsymbol{r}_u$ in the set Y using Pearson correlation:

$$w_{a,u} = \sum_{Y_{a,u}} \frac{(r_{a,i} - \underline{r_a})(r_{u,i} - \underline{r_u})}{\sigma_a \sigma_u \, | Y_{a,u} |}$$

$$\underline{r_u} = \sum_{Y_{a,u}} \frac{r_{u,k}}{| Y_{a,u} |}$$

$$\sigma_u^2 = \sum_{Y_{a,u}} \frac{(r_{u,k} - \underline{r_u})^2}{| Y_{a,u} |}$$

$$Y_{a,u} = \{ k \mid k \in M \cup r_{a,k} \neq \Theta \cup r_{u,k} \neq \Theta \}$$

Example 3.

We continue our running example and apply *ComputeCorrelationSimilarity()* on the two random partitions from Example 2. In each partition, the user's mean item-rating $\underline{r_u}$ and the standard deviation σ_u are computed from the items that both users have rated in common. The correlation between Sammy and Dylan for example is computed only over the movies, *Starship Trooper*, *Sleepless in Seattle*, and *MI-2*; we withhold Sammy's ratings for *Matrix* and *Titanic* for testing. The correlation between Dylan and Mathew is computed over all movies since their ratings for all five movies is available for training. The similarity matrix for each of the partitions is summarized in the table below.

		Correlation		
		Partition #1		
		Beatrice	Gum-Fat	Basil
Users	Beatrice	1	-0.92	-1
	Gum-Fat	-0.92	1	1
	Basil	-1	1	1

		Correlation		
		Partition #2		
		Sammy	Dylan	Mathew
Users	Sammy	1	1	-0.87
	Dylan	1	1	0.21
	Mathew	-0.87	0.21	1

Figure 6: Pair-wise similarity weights between members of the same random partition.

□

Theorem 1. The subroutine *ComputeCorrelationSimilarity()* has O(n^2/k) complexity.

Proof. The cost of computing the pair-wise similarity between members of the same partition is O($(n/k)^2$), where n is the data set size and k is the number of partitions. The complexity of computing the similarity weights for all k partitions is then O(n^2/k). □

3.2.4 Prediction Phase

We approximate the mapping of advisors' ratings to a user recommendation with a linear function. Recently published linear mappings include weighted average [SM95], weighted deviations from mean [RIS+94], and most probable rating [PHL00]. The latter approach is not applicable since we have not selected PD as our similarity metric. We now provide the rationale for selecting the weighted deviations from mean approach over the weighted average approach.

We postulate that for a majority of users the absolute rating scale is not meaningful. Rather, users rate items on a relative scale - they rate an item in context with the ratings they have previously assigned to other items. Mathew for example, likes *MI-2* and likes *Matrix* even more and therefore assigns scores of 3 and 4, respectively. His rating does not mean that *MI-2* is a mediocre movie, but that *Matrix* is better. We further postulate that users are ambivalent about a majority of items, like or dislike a small number of items, and are passionate about even fewer items. Given these hypothesises, we expect that a user's rating distribution will resemble a bell curve with a mean that is not necessarily located at the rating scale's median value. This hypothesises is supported by a plot of the rating distribution for the EachMovie[5] database in Figure 7. The rating mean is not at the scale median (between 2 and 3) - supporting our first

[5] The EachMovie movie recommendation service ran for a period of 18 months starting in 1996 and gathered almost 3 million ratings from 73,000 members.

hypothesis. The rating distribution is not uniform, but shows a peak about which the votes decline on either side - supporting our second hypothesis. The subroutine *ComputeDeviationFromMeanPrediction()* generates predictions that are distributed in a bell curve about the user's mean rating $\underline{r_u}$. It aggregates the weighted deviations of all of the advisors' rating from their mean. The weights w_{ij} are the pair-wise correlation coefficients computed from the subroutine *ComputeCorrelationSimilarity()*.

Subroutine 2. *ComputeDeviationFromMeanPrediction(S)*

Input: *S* is the test set and is the database of all user "no-rating" item vectors; each vector is the set of items for which the user has yet to rate and for which we would like to produce predictions.

Output: Each element of each vector of *S* is mapped into a rating score or "no rating", *Θ*.

Method: A prediction $p_{a,i}$ for the active user *a* and item *i* is computed from the weighted sum of deviations from the mean $\underline{r_u}$ of all its advisors. The weights $w_{a,u}$ are computed from the method *ComputeCorrelationSimilarity(..)*.

$$p_{a,i} = \underline{r_a} + \tfrac{1}{\alpha} \sum_{Y_{a,u}} (r_{u,i} - \underline{r_u}) \cdot w_{a,u}$$

$$\alpha = \sum_{Y_{a,u}} \frac{|w_{a,u}|}{|Y_{a,u}|}$$

$$Y_{a,u} = \{u \mid w_{a,u} \neq 0\}$$

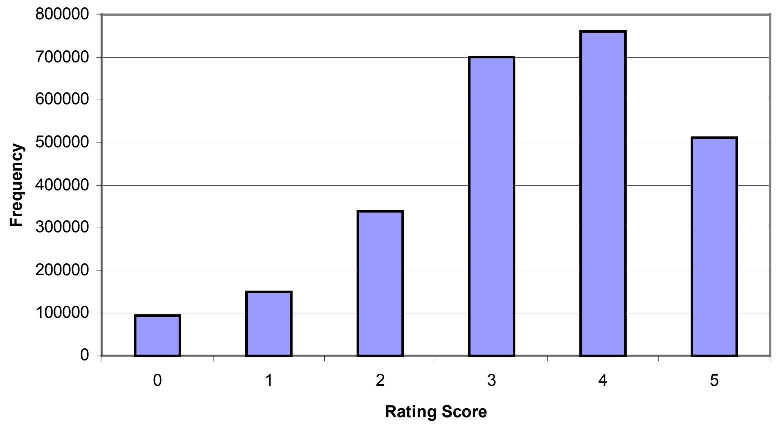


Figure 7: The EachMovie rating distribution.

Example 4.

In this example, we generate Sammy's and Basil's predictions for the movies *Matrix* and *Titanic* using the subroutine *ComputeDeviationFromMeanPrediction()*. The similarity weights w_{ij} are obtained from the tables in Example 3. The results of the calculation are summarized in the table below. The mean absolute difference between the prediction and the actual rating is shown in the column labelled MAE.

		Prediction		Actual		MAE
		Matrix	Titanic	Matrix	Titanic	
Users	Sammy	3.6	2.8	3	4	0.9
	Basil	4.6	4.1	4	5	0.75

$$P_{Sammy,Matrix} = \underline{r_{Sammy}} + \begin{Bmatrix} (r_{Dylan,Matrix} - \underline{r_{Dylan}}) \cdot w_{Sammy,Dylan} + \\ (r_{Mathew,Matrix} - \underline{r_{Mathew}}) \cdot w_{Sammy,Mathew} \end{Bmatrix} \cdot \frac{1}{|w_{Sammy,Dylan}| + |w_{Sammy,Mathew}|}$$

$$= 3.3 + \{(3-3.4)\cdot 1 + (2-3.2)\cdot(-0.87)\}/(1+0.87)$$
$$= 3.6$$

□

Theorem 2. The prediction step has O(*l/k*) complexity.

Proof. RandNeighCorr partitions a data set of *l* users into *k* partitions of approximately *l/k* size. A prediction is generated by aggregating the weights of all members of the partition and is therefore O(*l/k*). □

3.2.5 Time Complexity

RandNeighCorr is the fastest partitioned collaborative filter that we study in this work with a time complexity of O($n\beta$) + O(l), where n is the data set size, l is the number of users, and β is the partition size. In this section we discuss how each of the phases in *RandNeighCorr* contribute to this total complexity.

By Corollary 1, Theorem 1, and Theorem 2 the accumulated cost of partitioning the data set, training the collaborative filter, and generating predictions is:

$$O(l) + O(n^2 / k) + O(l / k)$$

(8)

Theorem 3. The execution time for *RandNeighCorr* is O(n) for constant partition size β.

Proof. Let the partition size β be given by n/k. Substituting for β in (8) we obtain *RandNeighCorr*'s complexity which is linear in data set size n: $O(n\beta) + O(l) + O(l/k)$ □

The constraint β is equivalent to fixing the number of item-ratings per partition and we therefore call β the partition size. By Theorem 3, we know that the execution time for *RandNeighCorr* can be linearized for constant values of β. In the next chapter we present our performance study into *RandNeighCorr* and demonstrate the effect of β on its performance.

3.2.6 Space Complexity

RandNeighCorr has a space complexity of $O(l^2/k)$, where l is the number of users and k is the number of partitions.

In the first phase all of the users are distributed among k partitions. This requires at most the storage of a 4-byte partition identifier per user; for the entire data set of users this operation requires $O(l)$ disk space. In the second phase, the pair-wise similarity coefficients between all members of a partition are computed and saved. If there are k partitions, the average number of users in a partition is l/k and the space complexity of computing pair-wise similarity coefficients for a partition is $O(l^2/k^2)$ and for all k partitions is $O(l^2/k)$. Therefore, the total space complexity of *RandNeighCorr* is:

$$O(l) + O(l^2/k)$$

3.2.7 *RandNeighCorr*'s Accuracy

The accuracy of a collaborative filter depends upon its ability to locate good advisors. This obvious observation is supported by the experiment summarized in Table 5. In the "All" column all advisors are used in the prediction, while in the "< 0.1" column only advisors with correlation coefficients less than 0.1 are used for prediction. As expected, using only advisor who are poorly correlated worsens the accuracy of predictions.

MAE	
All	< 0.1
0.87	0.97

Table 5: A comparison of *CorrCF*'s MAE with all correlates and when only low correlates (less than 0.1 correlation) are used in the prediction. A lower score indicates higher accuracy.

To improve the execution time, *RandNeighCorr* limits the neighbourhood over which advisors are sought. Intuitively we expect that the accuracy of *RandNeighCorr* should be affected. The following theorems confirm this intuition and the corollary proves the result that *RandNeighCorr*'s accuracy diminishes with increasing data set size. This result is somewhat surprising, as a collaborative filter should improve in accuracy as more data is available for it to train on.

Theorem 4. The probability of locating the f globally best advisors in a database of l users, which has been partitioned into k partitions of approximately equal size, is $O(1/k^f)$.

Proof. The probability that a user's most similar advisor is located in his partition is equal to the probability of being located in any of the k-1 other partitions and is therefore $1/k$. Since users are distributed into each partition randomly and the partitions are independent, the probability that each of the second and subsequent f-1 best advisors is also collocated is also $1/k$. The total probability therefore of locating the f globally best advisors is the product of these probabilities: $1/k^f$. □

Theorem 5. The probability of locating the f globally best advisors in a partition is $O(\beta^f n^{-f})$.

Proof. RandNeighCorr guarantees its linear execution time by the constancy of the partition size $\beta = n/k$. As the data set size n increases, the number of partitions k must also increase to maintain □'s constancy. Substituting for β in Theorem 4 we have the probability of locating the f globally best advisors in a partition: $O(\beta^f n^{-f})$. □

Corollary 2. The accuracy of *RandNeighCorr* diminishes with increasing data set size.

Proof. The accuracy of a collaborative filter depends upon its ability to locate the f best advisors to draw recommendations from. By Theorem 5 we know that as the data set size increases, the probability of locating these advisors in a partition is $O(n^{-f})$. □

3.3 *KMeansCorr*

3.3.1 Overview

KMeansCorr is a refinement on the *RandNeighCorr* algorithm. The random assignment strategy of phase I is replaced by a clustering algorithm that subdivides the database into partitions of correlated users. There exists a collection of clustering algorithms that we can choose from [KR89] [EKS+96] [ZRL96] [GRS98] [AGG+98] [ACW+99]. We make our selection based on the following criteria: the clustering algorithm must have quadratic time complexity or better, be applicable to high dimensional sparse data and be straightforward to implement. The first criterion is necessary since the *raison d'être* of this thesis is on the development of a faster collaborative filter; a clustering algorithm that took longer than quadratic time to complete would negate the very benefit that was intended. The second criterion is a necessary applicability condition: each item will be treated as a dimension along which a clustering is sought and the number of users who may have sampled any particular item is usually quite small. The average user of the EachMovie recommendation service [EM97] for example, has sampled only 10 movie titles out of a possible 1628. The third criterion, though not necessary, was deemed a desirable property due to time constraints for the completion of this work.

Several of the recent clustering algorithms [ZRL96] [GRS98] [AGG+98] [ACW+99] were designed specifically to handle high dimensional data in very large databases. These algorithms are disk-based and consequently very efficient when the database is larger than memory. However, this work assumes (and argues in 3.1.2) that all of the rating data will fit into memory; the *KMeans* algorithm [KR89] is a very fast in-memory clustering algorithm and with some modification can serve our purposes. We call this extended clustering algorithm *KMeans*$^+$ and describe it in detail in Algorithm 2. The CF algorithm we call *KMeansCorr* consists of replacing the partition phase (Phase I) of Algorithm 1 with *KMeans*$^+$.

It should be noted that our purpose in selecting a clustering algorithm for phase I is neither to locate nor to identify the cluster structure in the data set. We partition the data because we want to improve the execution time of the collaborative filter. In section 3.2 we show that a random partitioning of the data resulted in linear time complexity, but with an increasing loss in accuracy with data set size. *KMeansCorr* attempts to remedy the deterioration in accuracy by populating each partition with users that are highly correlated. Therefore, the purpose of inserting a clustering algorithm into Phase I is to seek a 'good' partitioning of the data such that execution time is linearized. Improving accuracy is a secondary objective. We are not using the clustering algorithm to identify cluster structures in the data.

3.3.2 *KMeans*

KMeans is a straightforward and fast in-memory clustering algorithm. Selecting *k* random points as temporary centres initializes the algorithm. It then iterates over the steps of assigning points to temporary centres and computing new centres until either the change in computed centres falls below a threshold or the maximum number of iterations is exceeded. *KMeans* has a time complexity that is proportional to *ktn*, where *k* is the number of clusters, *t* is the maximum number of iterations and *n* is the number of data points. Consequently, *KMeans* has linear time complexity with the number of data points.

KMeans has a number of limitations. The seeding of the algorithm with *k* initial starting centres is problematic - given inappropriate starting centres the algorithm can be trapped by locally optimal solutions. Secondly, the initial centres are chosen by random selection; consequently, each run of the *KMeans* algorithm may yield different clusters. Furthermore, *KMeans* cannot handle missing data. A distance calculation between a point and a candidate cluster centre can only be carried out if both vectors have fully specified dimensions. Finally, *KMeans* is susceptible to outliers; the algorithm has no way of detecting and removing outliers from its calculation of the cluster centre. Figure 8 illustrates some of these conditions. In the following subsections we discuss how *KMeans*$^+$ overcomes some of these limitations.

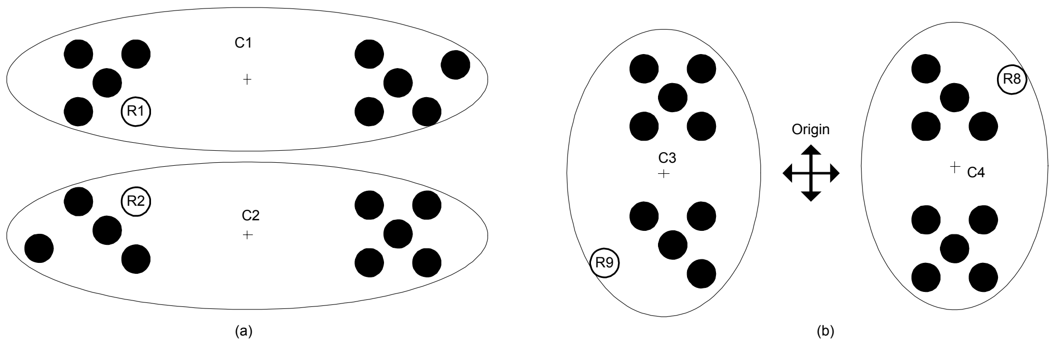


Figure 8: Two different selections for the initial centres yield two different sets of clusters. Selecting R1 and R2 as the seeds results in cluster C1 and C2, while selecting seeds R8 and R9 results in the globally optimal clusters C3 and C4.

3.3.3 *KMeans*$^{+}$

3.3.3.1 Overview

We denote the *KMeans* clustering algorithm with the extensions described below as *KMeans*$^{+}$. *KMeans*$^{+}$ can handle sparse data and uses correlation as the distance metric between users.

The *KMeans*$^{+}$ algorithm has a two-step initialization phase: it first computes a default vector that is part of the strategy for dealing with sparse data (Step 1) and then selects k temporary centres to seed the algorithm (Step 2). It then iterates over the following steps. *KMeans*$^{+}$ scans over the data set and assigns each vector to the cluster centre that it is closest to. If a vector is missing a term, the default vector supplies it (Step 3). Once all the vectors are assigned, the centroid of each cluster is computed from the average coordinate of all its members and a global counter is incremented (Step 4-5). *KMeans*$^{+}$ iterates over these steps (Step 3-5) until the maximum number of iterations t has been exceeded or the cluster centres have stabilized (Step 6-7).

In the following subsections we discuss each phase of the *KMeans*$^{+}$ algorithm in detail. The *KMeans*$^{+}$ algorithm differs from *KMeans* in its missing value replacement strategy and its seed selection strategy (Step 1-3). *KMeans* does not have a way to deal with missing data and it selects seeds by random selection.

Algorithm 2. ***KMeans+(Y, k, t, minChange)***

Input: *Y* is the training set and is the database of all user item-rating vectors. *k* is the number of partitions to create and k << |*Y*|. *minChange* is the change in the cluster centres' position below which we assume the clustering algorithm has converged while *t* is the maximum number of iterations we allow before terminating the clustering algorithm.

Output: *k* partitions that are characterized by their centroids: $c_1, c_2, .., c_k$

Method: The partitions are computed using the following procedure:

```
1. Compute the default vector $\boldsymbol{r_o}$ of $Y$ as the centroid:
       $\boldsymbol{r_o} = \Sigma \boldsymbol{r_i}/|Y| \quad , i \in Y$
2. Select $k$-mutually-well-separated centres: $c_1, c_2, .. c_k$
3. Assign each vector in $Y$ to the cluster, $C_j$ whose centre $c_j$ is most correlated. Use the default
   vector, $\boldsymbol{r_o}$, for missing value replacement.
4. Compute the centre of each cluster from the average of its members.
       $\boldsymbol{c_j} = \Sigma \boldsymbol{r_i}/|C_j| \quad , i \in C_j$
5. iterations++
6. If iterations > $t$ then goto Step 3
7. If the change in all of the clusters' position relative to the last iteration is above the threshold,
   minChange, then goto Step 3
       $d(\boldsymbol{c_j}, \boldsymbol{c_j}^{prev}) < minChange \qquad 1 \le j \le k$
8. STOP.
```

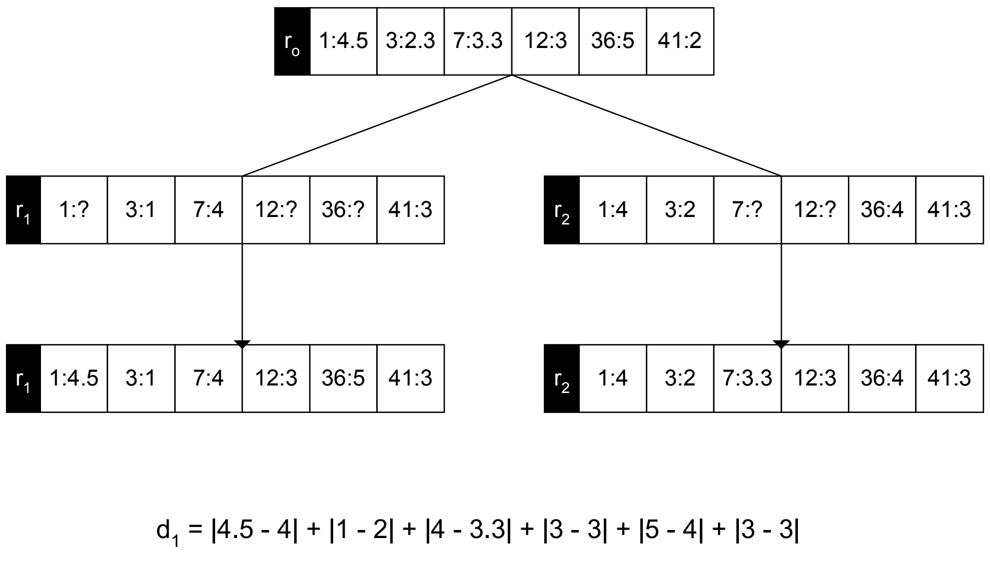
3.3.3.2 The Missing Value Replacement Strategy

The item-rating vectors in CF applications are very sparse with many terms that have the "no rating" value, Θ. This presents a problem to our clustering algorithm since a distance metric can only be computed over numeric values. We propose to handle this problem by using a default vector for missing value replacement (Step 3). In Figure 9 we show a Euclidean distance calculation between the item-rating vectors $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$ with missing values replaced by the default vector $\boldsymbol{r_o}$. Each cell consists of an item identifier and rating score pair; the "?" indicates a missing value. User $\boldsymbol{r_1}$ for example rates item 3 with a score 1 and has not rated items 1, 12 and 36.

A default vector is used to replace missing values in the user's rating vector. It therefore represents a guess at how the user will rate an item once he has seen it. This replacement strategy is necessary for our clustering algorithm, which requires fully specified vectors.

How do we compute a default vector? By definition, a collaborative filter provides us with these missing ratings since its very purpose is to compute predictions for items that the user has yet to see. Selecting a personalized recommender for this task is circuitous; we are using a clustering algorithm to speed up a collaborative filter but we are using a collaborative filter to initialize the clustering algorithm. Can any performance gain be realized with this arrangement? Personalized recommenders have at best quadratic complexity and introducing them into our algorithm would create a performance bottleneck. Selecting a non-personalized recommender is a different story however. Non-personalized recommenders have the potential to benefit the clustering algorithm since they are very fast.

We select the population average recommender, "*PopAvg*" for brevity, to generate the default vector. *PopAvg* generates a value for each term in the default vector by computing the average rating of an item. The default vector as computed by *PopAvg* is equivalently the cluster centroid if we view the data set itself as a cluster. *PopAvg* is a fast recommender and can compute the default vector with a single scan over the data set. The *PopAvg* recommender is used to compute the default vector in Step 1 of the *KMeans*$^{+}$ algorithm.



r_o 1:4.5 3:2.3 7:3.3 12:3 36:5 41:2
r_1 1:? 3:1 7:4 12:? 36:? 41:3
r_2 1:4 3:2 7:? 12:? 36:4 41:3
r_1 1:4.5 3:1 7:4 12:3 36:5 41:3
r_2 1:4 3:2 7:3.3 12:3 36:4 41:3
$d_1 = |4.5 - 4| + |1 - 2| + |4 - 3.3| + |3 - 3| + |5 - 4| + |3 - 3|$

Figure 9: The default vector provides missing values.

Example 5.

The default vector for the video database is computed using the *PopAvg* recommender and summarized in the table below.

Titles				
Starship Trooper (A)	Sleepless in Seattle (R)	MI-2 (A)	Matrix (A)	Titanic (R)
3.7	3	3.5	3	3.5

Figure 10: The default vector is computed for the video database using the *PopAvg* recommender.

□

Theorem 6. The default vector is computed in O(n).

Proof. *PopAvg* computes the default vector. *PopAvg* requires a single scan of the database to accumulate the average rating for each item and therefore the default vector is generated in O(n) where n is the size of the data set. □

3.3.3.3 The Seed Selection Strategy

The selection of the initial seeds for the *KMeans* algorithm is a subtle issue. Selecting poor seeds can result in the algorithm gravitating to locally optimal solutions. One approach to overcoming this problem is to execute the *KMeans* algorithm several times (selecting a new random set of seeds during each execution) and choosing the best clusters from the result sets. Another approach involves executing *KMeans* on a sub-sample of the dataset [BP98]. However, both of these approaches only improve the probability of locating the globally optimal solution. They do not guarantee it.

Fortunately, in this work a globally optimal clustering is not necessary. We use the *KMeans*$^+$ algorithm not as a means to discover the cluster structure in the data set, but as a means to create partitions of correlated users.

Furthermore, we want the *KMeans*$^+$ algorithm to create partitions of approximately equal size, otherwise the partitioned collaborative filter will not scale linearly. If for example k-1 clusters of size 1 and 1 cluster of size n-k+1 are created, the cost of training the collaborative filter will be O(n-k+1)^2 ~ O(n^2). On the other hand, if k clusters each of approximately n/k size are created, the cost of training a single partition is O($(n/k)^2$ and the cost of training all k partitioning is O(n/k^2) which is linear with the data set size n.

KMeans$^+$ selects its seeds using a strategy that we call k-mutually-well-separated centres (Step 2). This procedure picks seeds, which under certain conditions (see Proposition 1), increases the probability that *KMeans*$^+$ will create clusters of approximately equal size. The procedure we follow to achieve this is as follows. First, from the database of user item-rating vectors Y, select the vector that is farthest from the rating centroid (i.e. the default vector) and add it to the set P. Until $|P|$ is k, iteratively select vectors from the set Y-P whose sum of distances to all members of P is greatest. In Figure 8, the principle of k-mutually-well-separated centres would result in the selection of the points ***R8*** and ***R9***. Notice that this selection of seeds results in the *KMeans*$^+$ algorithm converging to the globally optimal clusters C_3 and C_4 in Figure 8.

This approach to seeding the *KMeans*$^+$ algorithm may be criticized for increasing its susceptibility to outliers since outliers tend to be points that are distant from other points [GRS98]. However, the partitioning phase of *KMeansCorr* is intended to find neighbourhoods of approximately similar users. The final computation for similarity is carried out in Phase II with a pair-wise evaluation. Consequently, a partition that includes some outliers merely results in the cost of computing their similarity to other users, but has negligible effect on the prediction since these users will have low similarity scores.

Example 6.

In this example, we apply the principle of k-mutually separated centres to select two temporary seeds from the video database. The default vector, as computed in Example 5, is used to replace the missing values in the training data for Sammy and Basil's ratings of *Matrix* and *Titanic*.

Gum-Fat is the furthest from the origin with a value of 8.54. We next compute the distance from Gum-Fat to the other members of the database. Beatrice is the furthest from Gum-Fat and therefore we selected her as the second seed.

If we need to select three seeds, the next candidate will be selected on the basis that his/her total distance to Gum-Fat and Beatrice is the greatest.

		Titles					Distance	
		Starship Trooper (A)	Sleepless in Seattle (R)	MI-2 (A)	Matrix (A)	Titanic (R)	to Origin	to Gum-Fat
Users	Sammy	3	4	3	3	3.5	7.43	2.00
	Beatrice	3	4	3	1	1	6.00	3.46
	Dylan	3	4	3	3	4	7.68	2.00
	Mathew	4	2	3	4	5	8.37	1.41
	Gum-Fat	4	3	4	4	4	8.54	0.00
	Basil	5	1	5	3	3.5	8.50	2.65

Table 6: The default vector is computed for the video database using the PopAvg recommender.

□

Theorem 7. The cost of locating k-mutually well-separated centres is $O(k^2n)$.

Proof. The selection of the ith seed requires a single scan over the database (of size $n-i$) and i-1 comparisons with the other members already in the set P. By induction, the cost to select k seeds is at most $k(k-1)n \sim O(k^2n)$. □

Proposition 1. When k=2 and the data distribution is Gaussian, the principle of "k-mutually well separated centres" results in two clusters of approximately equal size.

Proof. For k=2 the two seeds form the diameter of the rating-space volume. Since the data has a Gaussian distribution, each seed lies at the opposite 'tails' of the distribution and half of the data will be closest to each. As a result of these conditions the *KMeans*$^+$ algorithm will stabilize at two clusters of approximately equal size.

3.3.3.4 The Distance metric

The *KMeans* algorithm assigns each point to the cluster centre that it is closest to. However, we have yet to specify what metric will be used to compute distance. Metrics that have been used for *KMeans* clustering include: Manhattan distance, Euclidean distance, and the other Lp metrics [KR89]. For our application, we want neighbourhoods of users who are highly similar in rating behaviour. Since phase II of *KMeansCorr* uses correlation to compute similarity (i.e., *ComputeCorrelationSimilarity(..)*), we will also use correlation as the distance metric for the clustering (Step 3) to obtain partitions of highly correlated users.

Unlike the Lp metrics where a smaller value denotes proximity, larger values denote proximity in correlation space. In correlation space the user is "closest" to the cluster centre to which his correlation score is the highest. The correlation between ratings can range from –1 to 1. A value of 1 denotes that the ratings are perfectly correlated and two users occupy the same point in correlation space. A value of –1 denotes anti-correlation; the users' rating behaviour are opposite and the two users are maximally distant.

The *KMeans*$^+$ algorithm therefore assigns each user to the cluster centre to which his or her correlation coefficient is the maximum.

Example 7.

In this example, we demonstrate the partitioning of the video database into two partitions using the *KMeans*$^+$ algorithm. The two-step initialization of the algorithm is shown in Example 5 and Example 6 where the default vector and the cluster seeds are selected, respectively. The training set is shown in Table 6 where the default vector has been used to replace the missing values for Sammy and Basil (the shaded cells). The cluster seeds are shown in the table entries for c1 and c2 for the set of tables labelled Iteration 1.

Iteration 1

		Titles				
		Starship	Sleepless	MI-2	Matrix	Titanic
centres	c1	4.0	3.0	4.0	4.0	3.0
	c2	3.0	4.0	3.0	1.0	1.0

		Correlation to	
		c1	c2
Users	Sammy	-0.9	0.4
	Beatrice	-0.1	1.0
	Dylan	-1.0	0.1
	Mathew	-0.2	-0.3
	Gum-Fat	1.0	-0.1
	Basil	0.7	-0.2

In the first iteration, the correlation between each user and the cluster seeds c1 and c2 are computed. We see that Sammy, Beatrice, and Dylan are closest to the cluster centre c1, while Mathew, Gum-Fat, and Basil are closest to c2. The new centroid of each cluster is computed from its members, yielding the new centres c3 and c4 (shown in the Iteration 2 tables).

In the second iteration, the correlation calculation yields two clusters with the same members as in the previous. We can therefore terminate *KMeans*$^+$ as the cluster centres have stabilized. The *KMeans*$^+$ algorithm therefore created two clusters with members {Sammy, Beatrice, Dylan} and {Mathew, Gum-Fat, Basil} respectively.

Iteration 2

		Titles				
		Starship	Sleepless	MI-2	Matrix	Titanic
centres	c3	4.3	2.0	4.0	3.0	3.8
	c4	3.0	4.0	3.0	2.3	3.3

		Correlation to	
		c3	c4
Users	Sammy	-0.7	0.9
	Beatrice	-0.3	0.7
	Dylan	-0.5	0.8
	Mathew	0.7	0.0
	Gum-Fat	0.5	-0.8
	Basil	1.0	-0.5

□

Theorem 8. *KMeans*$^+$ has time complexity of O(n) + O(k^2n) + O(ktn).

Proof. By Theorem 6 the cost to compute the default vector is O(n), where n is the data set size. By Theorem 7 the seed selection strategy has complexity O(k^2n) where k is the number of seeds to select. During an iteration (Step 3-6) each point must be compared against k temporary centres and a complete iteration therefore requires kn comparisons. The algorithm is bounded by the limit of t iterations and consequently the complexity for the iterative phase is ktn. The total complexity is linear with the size of the data set: O(n) + O(k^2n) + O(ktn) □

3.3.4 Time Complexity

KMeansCorr has complexity of O(n^2), where n is the data set size. Let $\underline{m}$ be the average number of item-ratings per user and l be the number of users in the data set, then define n as the size of the data set given by $\underline{ml}$.

KMeansCorr partitions the data set in Phase I using the *KMeans*$^+$ algorithm. It trains the collaborative filter in Phase II by computing pair-wise correlation coefficients with the subroutine *ComputeCorrelationSimilarity(..)* and generates predictions in Phase III with the subroutine *ComputeDeviationFromMeanPrediction(..).* Theorem 8, Theorem 1, and Theorem 2 give the accumulated cost of these phases:

$$O(n) + O(k^2n) + O(ktn) + O(n^2/k) + O(l/k) \tag{9}$$

Theorem 9. *KMeansCorr* has time complexity of O(n^2).

Proof. Imposing the constraint that $n/k = \beta$ is a constant, the training and prediction phase of *KMeansCorr* can be linearized. However, substituting for β in (9) we find that the partitioning phase's complexity is cubic under this constraint: O(n) + O(n^3/β^2) + O($tn^2\beta$). Therefore the best achievable time complexity is O(n^2) and is achieved by allowing the partitioning phase to complete in linear time and executing the training phase in quadratic time. □

KMeans$^+$'s complexity is dependant upon both the number of item-ratings n and the number of partitions k. The best achievable complexity is O(n^2) and is obtained by fixing the number of partitions k. This results in the partitioning phase completing in linear time, but the training phase completing in O(n^2) time. Despite the disappointing result this approach is instructive and we draw a number of lessons from it:

- The training and prediction phase consisting of *ComputeCorrelationSimilarity(..)* and *ComputeDeviationFromMeanPrediction(..)* can be made linear if the partition size β is kept constant.
- The partitioning phase consisting of *KMeans*$^+$ can be made linear if the number of partitions k can be kept constant.

The *KMeansCorr* algorithm has quadratic complexity at best, because it cannot optimize the partitioning and prediction phase independently. In section 3.4, we describe a new algorithm, *RecTree*, that builds upon *KMeans*$^+$ and is linear in complexity.

3.3.5 Space Complexity

KMeans$^+$ has a space complexity of O(l) + O(km), where l is the number of users in the data set, k is the number of partitions and m is the size of the item set, M (i.e.: $m \equiv |M|$).

During each iteration, users are assigned to the cluster centre that is closest. This operation requires at most 4-bytes per user to store the current cluster identifier. Each iteration of the algorithm can overwrite this identifier since a history of cluster identifiers is not required. Storing the cluster identifiers for all l users therefore requires O(l).

Each of the cluster centroids may span the entire item space and therefore each centroid requires at worst O(m) storage. For k partitions the storage for all centroids is O(km).

3.4 *RecTree*

3.4.1 Overview

RecTree is the acronym for a new data structure and collaborative filtering algorithm called the RECommendation Tree. The *RecTree* algorithm partitions the data in phase I by recursively calling *KMeans*$^{+}$ to split the data set into child clusters. The chain of intermediate clusters leading from the initial data set to the final partitioning is maintained in the *RecTree* data structure, which resembles an unbalanced binary tree. The collaborative filter is trained in Phase II by computing the pair-wise similarity coefficient within each leaf partition using a more accurate similarity metric that we call *correlation*$^{+}$. The internal nodes are processed by the collaborative filter by computing the rating centroid based on the item-vectors in its sub-tree. The *RecTree* generates a prediction in Phase III by employing a dual strategy. If the user is located in a leaf partition with a *sufficient* number of neighbours, taking the weighted deviation from the mean generates a prediction. If the user is located in a small neighbourhood, *RecTree* generates a prediction by returning the neighbourhood's average rating.

A prediction request is made by passing a message directly to a tree node for processing. A recommendation message takes the form of 2 integer identifiers and a double representing the prediction. The *RecTree* responds to a message by filling in the prediction field in the message.

UserID: integer	ItemID: integer	Prediction: double

In the subsections below we describe in detail how to construct the *RecTree* and then how to query it for recommendations.

3.4.2 Constructing the *RecTree*

The *ConstructRecTree(..)* routine builds the *RecTree* data structure by recursively splitting the data set *Y* into child clusters until stopping conditions are met. *ConstructRecTree(..)* interleaves Phase I and Phase II of our framework by partitioning the data and training the collaborative filter during the course of building the tree.

During each iteration a node is associated with the current data cluster and a bi-directional link is created to the predecessor (Step 2-3). The routine then classifies the node as an internal node, an outlier node, or a leaf node by checking the size of the cluster. If the node is classified as an internal node, the *KMeans*$^+$ splits the current data cluster into two child clusters and the *ConstructRecTree(..)* routine is recursively called to insert each of the child clusters into the tree (Step 7 & 9).

A node is classified as an outlier if it has fewer ratings than the threshold *outlierSize* and classified as a leaf if it has fewer ratings than the partition size parameter β (Step 4-5). A leaf node represents the final partitioning of the data and as such is ready to be trained on by a collaborative filter; the subroutine *ComputeCorrelationSimilarity+(..)* trains the collaborative filter on each leaf node (Step 5).

A node that is not classified as an outlier or a leaf is by definition an internal node. Internal nodes implement the secondary prediction strategy and train an average rating vector from all of the members of that node.

The *RecTree*'s construction is complete when the growth along every branch has been terminated. In addition to outlier and leaf nodes, the global iteration threshold *g* can also terminate the tree growth. If the *RecTree* has recursed more than *g* times then the recursion at every branch is terminated (Step 5).

The primary purpose of the *RecTree* data structure is to seek a 'good' partitioning of the data set such that the complexity of delivering recommendations is linearized. Improving the accuracy of predictions is a secondary objective. As with the *KMeansCorr* algorithm, the partitioning phase of *RecTree* is not intended to identify nor locate cluster structures.

In the following subsections we describe details of the algorithm and discuss the rationale for the design.

Algorithm 3. ***ConstructRecTree(parent, Y, β, g,outlierSize)***

Input: *parent* is the parent of the current node. *Y* is a database of user item-rating vectors. β is the threshold at which to stop subdividing a partition. *g* is the maximum number of times that this method will call itself before terminating. *outlierSize* is the threshold at which a leaf node is marked as an outlier node. o*utlierSize* < β.

Output: The *RecTree*.

Method: The *RecTree* is constructed by calling this method recursively. On the first invocation, the argument, *parent*, is NULL.

```
1. If parent is NULL then
       Set global_iterations = 0.
2. Build a node V and create a bi-directional link between V and parent.
3. Assign all members of Y to V.
4. If $|Y|$ < outlierSize then mark this leaf node as an outlier node; STOP.
5. If $|Y| \leq \beta$ OR (global_iterations > g) then
       ComputeCorrelationSimilarity$^+$(Y); STOP.
6. Compute the centroid, $c_Y$ of Y.
7. Call KMeans$^+$(Y, 2, t, minChange).
8. global_iterations++.
9. For each child cluster Y′:
       Call ConstructRecTree(V, Y′, $\beta$, g, outlierSize).
```

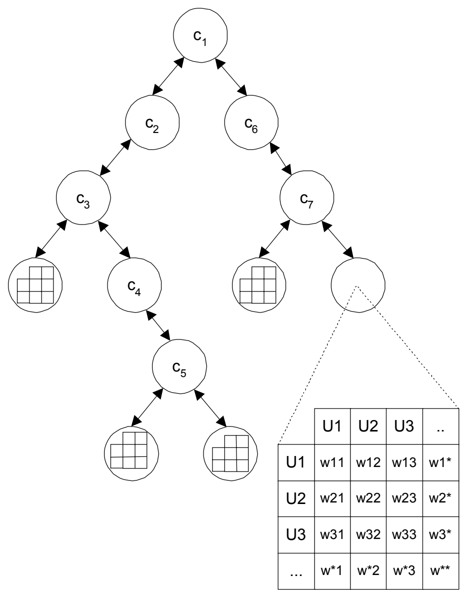


Figure 11: The *RecTree* data structure. Leaf nodes have a similarity matrix while each internal node maintains a rating centroid of its sub-tree. Each node has a bi-directional link with its parent.

3.4.2.1 Node Splitting

The *RecTree* is grown from the root downwards by node splitting. How do we decide when node splitting can be terminated? Hierarchical divisive clustering methods continue splitting a cluster until a quality threshold is reached. Metrics such as minimum, maximum, and average distance have been used to control the splitting [KR89]. However, each of these metrics may lead to clusters with a small membership. This condition is not appropriate to our application since the size of the partitions affects the coverage of the collaborative filter. A smaller neighbourhood implies fewer candidate advisors from which we can draw recommendations (See Figure 4). Therefore, in this work we investigate the use of a minimum neighbourhood size as a clustering control. A node that contains more item-ratings than the threshold, β, is split. Equivalently, a node that has fewer than β item-ratings is classified as a leaf node and growth along that branch is terminated (Step 5).

Once the decision to subdivide a node has been made, we employ the *KMeans*$^+$ algorithm described in 3.3.3 to cluster the data into two partitions (Step 7). *KMeans*$^+$ is suited to the task since it can handle sparse multidimensional data and has linear complexity with data set size n when the number of partitions, k, is held constant and independent of n. We satisfy this requirement in *ConstructRecTree(..)* by always calling *KMeans*$^+$ to split each internal node to exactly two clusters (Step 7). Each of the child clusters that results is added to the tree by a recursive call to the method *ConstructRecTree(..)* (Step 9).

Example 8.

In this example we demonstrate the branch growing steps of *ConstructRecTree(..)* on the video database. We assume that the partition size parameter, β, has been set to 15 item-ratings and ignore the *outlierSize* and *g* thresholds for the time being.

On the first invocation of *ConstructRecTree(..)*, a root node is created and associated with the entire video database of users (Step 2-3). The root node contains 26 item-ratings. This exceeds our partition size parameter β so we proceed to split the node into two child clusters by calling *KMeans*$^{+}$ (Step 5 & 7). The first invocation concludes with a root node and two child partitions shown as dashed circles.

ConstructRecTree(..) is then invoked on each of the child partitions. Each child partition is associated with a node and attached to the root through a bi-directional link. Each of these partitions has 13 item-ratings, the size parameter β is satisfied and the growth along each branch is terminated (Step 5).

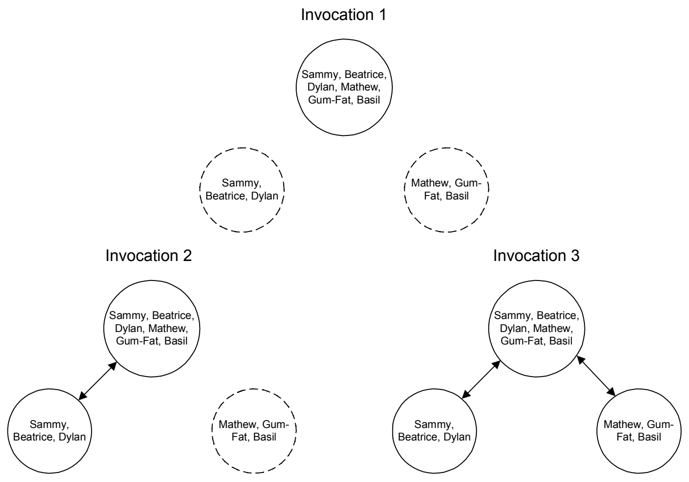


Figure 12: The *RecTree* branches after processing the video database with β=15 item-ratings.

□

3.4.2.2 The Maximum Iteration Limit, *g*

For some data distributions, the hierarchical clustering we propose may result in an over-partitioning of the data. Figure 13 illustrates one hypothetical data distribution that would be partitioned into *n-β* clusters of size 1 and one cluster of size *β*. Clearly the calculation of recommendations on this pathological data distribution and others similar to it would not benefit from any form of clustering. The small clusters would be useless since they have too few advisors from which to draw recommendations. A straightforward application of a collaborative filter on the un-partitioned data would have yielded the same recommendations in shorter time.

We therefore propose to avoid these problematic data distributions by limiting the maximum number of times that the data can be subdivided using an iteration threshold, *g*. Each time that a node is split we increment a global iteration counter (Step 8). We then test to see if the threshold *g* has been exceeded on each invocation of *ConstructRecTree(..)* (Step 5). *g* can equivalently be viewed as a limit on the maximum number of internal nodes in the *RecTree,* as demonstrated in Corollary 5.

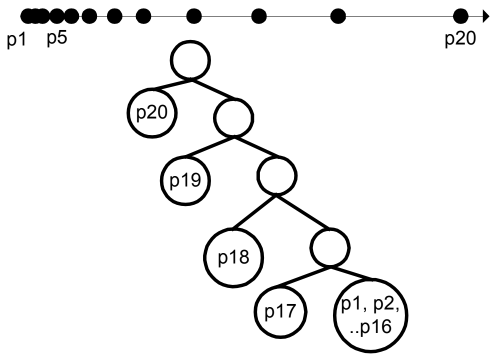


Figure 13: Building a *RecTree* on an exponential data distribution with *β* = 16 creates 4 clusters with one user each and 1 cluster with 16 users.

Lemma 1. The *RecTree* has at least *n/β* leaf nodes.

Proof. Assume that *g* is an arbitrarily large number. For a given data set with *n* item-ratings, the *RecTree* partitions the data until all nodes contains *β* or fewer item-ratings. When each partition has exactly *β* item-ratings, there are a minimum number of leaf nodes given by *n/β*. □

Corollary 3. If the *RecTree* has *n/β* leaf nodes then each leaf node has *β* item-ratings.

Proof. If a data set of *n* item-ratings is partitioned into clusters of β size then there are n/β clusters. □

Lemma 2. The *RecTree* has at most n-β+1 leaf nodes.
Proof. Assume that *g* is an arbitrarily large number. Suppose on each of i^{th} recursive calls to *ConstructRecTree(..)* the node V^2_{i-1} with n_{i-1} members is subdivided into the two child nodes V^1_i and V^2_i such that $|V^1_i| = 1$ and $|V^2_i| \equiv n_i = n_{i-1}-1$. Then *ConstructRecTree(..)* will be called n-β times – each recursion yielding another leaf node of size 1. On the n-β^{th} recursive call, two leaf nodes are created, one of size 1 and another of size β; the total number of leaf nodes is therefore n-β+1. If during any iteration, $|V^1_i| > 1$, then *ConstructRecTree(..)* will recurse fewer than n-β times and therefore yield fewer leaf nodes. Therefore *RecTree* has at most n-β+1 leaf nodes.□

Corollary 4. If the *RecTree* has n-β+1 leaf nodes then there are 2 leaf nodes whose total size is β and n-β nodes each of size 1.
Proof. By Lemma 2 there are at most 2 leaf nodes whose size is greater than 1 otherwise the tree has fewer than n-β+1 leaf nodes. □

Lemma 3. The *RecTree* has at most *g* internal nodes.
Proof. On each call to *ConstructRecTree(..)*, a leaf node is split into two child nodes. The former leaf node increases the population of internal nodes by 1. After the i^{th} recursive call to *ConstructRecTree*(..), there are i internal nodes. By induction, if the construction terminates after recursing at most *g* times, there are *g* internal nodes. □

Lemma 4. The *RecTree* has z+1 leaf nodes where z is the number of internal nodes.
Proof. On each recursive call to *ConstructRecTree*(..), a leaf node is split into two child nodes. The former leaf node becomes an internal node and the two child clusters are by definition leaf nodes and the population of leaf and internal nodes increases in total by 1. By induction, after the i^{th} recursive call to *ConstructRecTree*(..), the population of leaf and internal nodes has increased by i members. On the first invocation of *ConstructRecTree(..)* the tree has exactly one leaf node, the root. Therefore the total population of leaf nodes after the ith invocation is z+1 where z is the number of leaf nodes. □

Corollary 5. The *RecTree* has at most *g*+1 leaf nodes.
Proof. By Lemma 3 and Lemma 4 □

Theorem 10. The branches of the *RecTree* can be constructed with complexity O(*gtn*).
Proof. The branches of the *RecTree* are grown by recursively calling the *KMeans*$^+$ algorithm to split a node into two child clusters until the partition size parameter is satisfied or a maximum recursion threshold *g* is exceeded. By Theorem 8 the complexity for each invocation of *KMeans*$^+$ to create two child clusters is: O(*n*) + O(*4n*) + O(*2tn*), where *n* is the data set size and *t* is the maximum number of clustering iterations. *KMeans*$^+$ is invoked at most *g* times and consequently, the total complexity of inserting the branches is O(*5gn*) + O(*2gtn*) ~ O(*gtn*) since *t*>>1. □

3.4.2.3 The Tree Nodes

Every node that is added to the *RecTree* has a bi-directional link connecting to its predecessor and a reference to a cluster of data (Step 2-3). The *RecTree* filter is trained as each node is attached to the tree. As a consequence of *RecTree*'s dual recommendation strategy (see 3.4.3), the filter's training depends upon the node type. In the following three sections we discuss how each node type is used to train *RecTree*'s filter.

3.4.2.4 Training on the Leaf Nodes

RecTree's filter is trained on the leaf nodes by computing a similarity matrix between all members of the partition. The leaf nodes provide the coefficients by which advisors' recommendations can be weighted in aggregate to generate a personalized prediction. We propose to use an enhanced version of the correlation similarity measure, that we call correlation$^+$, to compute pair-wise user similarity.

The design of correlation$^+$ is motivated by a number of observations in regard to the recently proposed similarity metrics. In the statistical collaborative filters proposed by [RIS+94] and [SM95], pair-wise similarity is computed using only the ratings for items that both users have rated. If the item intersection set is small, then the proximity coefficients are poor measures of similarity as they are based on too few item-rating comparisons. Furthermore, an emphasis on similarity based on the intersection set neglects the global rating behaviour that is reflected in a user's entire rating history. The following example illustrates these problems.

Example 9.

Sammy and her friends have the rating history shown in the table below. A similarity measure that considers only items in the intersection set (*{Starship Trooper, Sleepless in Seattle, MI-2}*) would render Beatrice and Dylan indistinguishable and equally valuable advisors for Sammy. However, if we look at the entire rating history we note that Dylan appears more similar to

Sammy than Beatrice and Beatrice exhibits a greater rating volatility in comparison to Sammy and Dylan. This intuition is confirmed in the table columns that summarize the "global" average rating and standard deviation of ratings. These values are computed from all of the available rating data in the video training set.

		Titles						
		Starship Trooper (A)	Sleepless in Seattle (R)	MI-2 (A)	Matrix (A)	Titanic (R)	Average	σ
Users	Sammy	3	4	3	?	?	3.3	0.6
	Beatrice	3	4	3	1	1	2.4	1.3
	Dylan	3	4	3	3	4	3.4	0.5
	Mathew	4	2	3	4	5	3.6	1.1
	Gum-Fat	4	3	4	4	4	3.8	0.4
	Basil	5	1	5	?	?	3.7	2.3

Table 7. A calculation of the average rating and standard deviation over the longer history indicates that Dylan is more similar to Sammy than Beatrice.

□

We believe a collaborative filter that took into account global rating behaviour would be more accurate. It would locate advisors that were globally more similar to the active user rather than advisors that exhibited only local similarities. We incorporate this intuition into the design of correlation$^+$. This measure differs from the standard correlation-based similarity calculation in that the mean rating $\underline{r}$ and standard deviation σ are computed over the entire set of items rated by each user.

Extending the set of items over which a similarity measure is computed should improve the metric's accuracy in capturing user similarity. The *t*-statistic for the correlation coefficient is proportional to the square root of the sample set size [HM85] and therefore increasing the number of ratings increases the confidence of the correlation calculation. How do we extend the set over which the correlation is computed given that user ratings are sparse? As in the *KMeans*$^+$ algorithm we employ a missing value replacement strategy. We replace missing values in a user's item-rating vector by the values in a default vector.

The semantics of the default vector in this discussion are the same as those in the *KMeans*$^+$ clustering algorithm. Specifically, the default vector represents an estimate of the user's rating for items that he has yet to rate. Although it would be ideal to compute a personalized default vector for each user this approach is infeasible. Personalized recommenders have quadratic complexity at best and their operation would introduce a bottleneck into our linear recommender. We therefore select the non-personalized recommender, *PopAvg*, to generate the default vector. *PopAvg* provides the same prediction for every user and requires only a single scan to train. The default vector as generated by *PopAvg* is equivalently the rating centroid of the leaf partition.

The subroutine *ComputeCorrelationSimilarity*$^+$*(..)* computes a more accurate measure of user similarity with a higher confidence by using the global values of mean rating and standard deviation and by extending the intersection set with a missing value replacement strategy.

Subroutine 3. *ComputeCorrelationSimilarity*$^+$*(Y)*

Input: *Y* is a set of user item-rating vectors.

Output: A square matrix of pair-wise similarity coefficients between all members of *Y*.

Method: The similarity weights are computed in the following manner:

1. Compute the default vector $\boldsymbol{r_o}$ of *Y* as the centroid:

$$\boldsymbol{r_o} = \Sigma \boldsymbol{r_i} / |Y| \quad , i \in Y$$

2. For each user $\boldsymbol{r_a}$ compute the pair-wise similarity $w_{a,u}$ to each other member $\boldsymbol{r_u}$ using Pearson correlation over the set of items they **both** have rated ($Y_{a,u}$). Use the default vector to replace missing values in each user's rating vector. The average rating and standard deviation of each user is computed over the entire set of items that each user has rated (Y_u)

$$w_{a,u} = \sum_{Y_{a,u}} \frac{(r_{a,i} - \underline{r_a})(r_{u,i} - \underline{r_u})}{\sigma_a \sigma_u \, |Y_{a,u}|}$$

$$\underline{r_u} = \sum_{Y_u} \frac{r_{u,k}}{|Y_u|}$$

$$\sigma_u^2 = \sum_{Y_u} \frac{\left(r_{u,k} - \underline{r}_u\right)^2}{|Y_u|}$$

$$Y_u = \{k \mid k \in M \cup r_{u,k} \neq \Theta\}$$

$$Y_{a,u} = \{k \mid k \in M \cup r_{a,k} \neq \Theta \; or \; r_{u,k} \neq \Theta\}$$

Example 10.

The table below shows the correlation and correlation$^+$ coefficients for the video database. The correlation coefficients are computed only over the titles *Starship Trooper*, *Sleepless in Seattle*, and *MI-2.* Correlation$^+$ is computed over all five titles with the missing ratings for Sammy and Basil estimated by the default vector.

Beatrice and Dylan are indistinguishable to the correlation metric; they are perfectly correlated to Sammy. Correlation+ captures the global similarity between these advisors and Sammy and distinguishes them by assigning a significantly higher correlation coefficient to Dylan in comparison to Beatrice.

		Titles					Similarity to Sammy	
		Starship Trooper (A)	Sleepless in Seattle (R)	MI-2 (A)	Matrix (A)	Titanic (R)	Correlation	Correlation$^+$
Users	Sammy	3	4	3	3	3.75	1.00	1.00
	Beatrice	3	4	3	1	1	1.00	0.21
	Dylan	3	4	3	3	4	1.00	0.98
	Mathew	4	2	3	4	5	-0.87	-0.25
	Gum-Fat	4	3	4	4	4	-1.00	-0.75
	Basil	5	1	5	3	3.75	-1.00	-0.74

Table 8. A comparison of the correlation coefficients computed by ComputeCorrelationSimilarity$^+$ (denoted by column Correlation$^+$) with those by ComputeCorrelationSimilarity (denoted by column Correlation).

□

Proposition 2. The confidence of coefficients as computed by *ComputeCorrelationSimilarity*$^+$ is higher or equal to than those computed by *ComputeCorrelationSimilarity*.

Proof. The confidence of a correlation coefficient is given by the *t*-statistic [HM85]: $t = \dfrac{r\sqrt{q-2}}{\sqrt{1-r^2}}$

where r is the correlation coefficient and q is the number of samples. Since *ComputeCorrelationSimilarity*$^+$ computes the correlation over the set of items that both users have rated rather than just the items that they have rated in common, the confidence of the correlation coefficients generated by *ComputeCorrelationSimilarity*$^+$ is higher or equal to the confidence of coefficients generated by *ComputeCorrelationSimilarity.* □

3.4.2.5 Training on the Internal Nodes

RecTree's filter is trained on the internal nodes by computing the rating centroid of each node. The rating centroids represent the average voting preference for each item among the members of that neighbourhood (node). The rating centroids provide the data to support *RecTree*'s dual recommendation strategy.

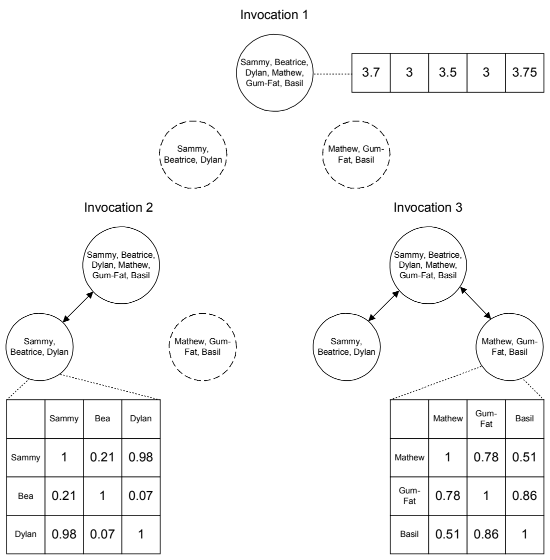
3.4.2.6 Training on the Outlier Nodes

In this collaborative filtering application we consider outliers as individuals whose preferences lie outside of the main of the *current* membership. However, as new users join the system or existing members rate additional items, these outlying individuals may find themselves among a group of like-minded members and thus cease to be outliers.

This perspective of outliers motivates us not to discard nor discount outliers, as customary in cluster analysis. Rather, we retain them in the leaf nodes of the *RecTree* until such time that enough like-minded individuals appear. [Guh84] recognizes outliers as data points that are distant from other data points. [EKS+96] considers a point as an outlier if the regional density is below a threshold. We modify this intuition for our CF application: *RecTree* marks a leaf node as an outlier node if it contains fewer than *outlierSize* item-ratings.

Example 11.

In this example, we demonstrate the training of the *RecTree* collaborative filter on the video database. During the first invocation, an internal node is created and the filter is trained by computing the rating centroid. In the second and third invocation, a leaf node is attached to the tree. The collaborative filter trains on the leaves by computing a similarity matrix for each leaf partition.



	Sammy	Bea	Dylan
Sammy	1	0.21	0.98
Bea	0.21	1	0.07
Dylan	0.98	0.07	1

	Mathew	Gum-Fat	Basil
Mathew	1	0.78	0.51
Gum-Fat	0.78	1	0.86
Basil	0.51	0.86	1

Figure 14: The RecTree's collaborative filter is trained as nodes are attached to the tree.

□

Theorem 11. Creating the leaf nodes in the *RecTree* is O($n\beta$).

Proof. The training of the collaborative filter on a leaf node requires the population of a similarity matrix, which is O(β^2) where β is the partition size. We estimate the complexity of training all the leaf nodes by examining the case when the minimum and maximum number of leaf nodes are created. By Lemma 1 and Corollary 3, the cost of training on the minimum number of leaf nodes is O($n\beta$). By Lemma 2 and Corollary 4, the cost of training on the maximum number of leaf nodes is O(n-β) + O(β^2). The cost of training on the leaf nodes is at therefore at worst O($n\beta$). □

3.4.3 Computing a Prediction

RecTree employs a dual strategy for generating predictions. When the size of the neighbourhood is sufficient, a prediction is computed by taking the weighted deviation from the mean. When the neighbourhood is small, a recommendation is computed by taking the rating average. *RecTree*'s dual strategy is based on the observation that the accuracy of personalized recommenders diminishes greatly when the neighbourhood is small [SKB+98] [GSK+99] [KM99a]. Furthermore, for very small neighbourhoods the accuracy of the non-personalized recommender *PopAvg* surpasses that of the personalized recommenders. By selecting an appropriate value for *outlierSize*, the threshold at which *RecTree* switches between its recommendation strategies can be controlled.

A prediction query message consisting of a user and item identifier is passed directly to a tree node for processing. How do we locate this node? We simply follow a pointer in the user data structure that links the user record with the tree node that he resides in. The manner in which each type of node processes a request is detailed below.

Leaf nodes implement *RecTree*'s primary prediction strategy. They compute a prediction by taking the weighted deviations from the advisors' mean. The subroutine *ComputeDeviationFromMeanPrediction(..)* described in 3.2 is used for this purpose. However the weights w_{ij} are those computed by the subroutine *ComputeSimilarityWeight*$^{+}$*(..)*. If a leaf cannot provide a prediction due to rating sparsity (i.e. the user's advisors have yet to rate the item of interest) then the request is delegated to the immediate parent (internal) node.

Outlier nodes by definition have small neighbourhoods and given our earlier discussion will generate poor predictions. An outlier node is used by *RecTree* to mark the transition to the second recommendation strategy. Any user that resides within an outlier node has recommendation requests bounced to the immediate parent (internal) node.

Internal nodes implement the secondary prediction strategy. Each internal node maintains an average item-rating vector that summarizes the rating behaviour of all the users in its sub-tree. An internal node can therefore answer a recommendation request by returning the average rating of its membership for that item.

RecTree's dual recommendation strategy is intended to overcome the deterioration in accuracy at low item-rating densities. It switches to the more robust non-personalized recommendation strategy of population averages when the size of the neighbourhood is small. When the neighbourhood is large, *RecTree* uses the prediction subroutine *ComputeDeviationFromMeanPrediction(..)*. This subroutine is more accurate with large neighbourhoods since it has a larger pool of candidates in which to seek good advisors. When the pool of candidates is small, the weighted recommendations are less accurate since the subroutine has fewer advisors on which to base a recommendation.

The outlierSize threshold controls the transition between the two recommendation strategies. If the *outlierSize* threshold is set to 0, a leaf node terminates every branch of the RecTree. The leaves handle all of the recommendation requests and *RecTree* degenerates to a single prediction strategy using *ComputeDeviationFromMeanPrediction(..)*. If outlierSize is set larger than the partition size parameter β then an outlier node terminates each branch of the RecTree. All recommendation requests are handled by returning the neighbourhood's average rating for the item of interest and *RecTree* degenerates to non-personalized recommendation via population averages.

Subroutine 4. *QueryRecTree(S)*

Input: *S* is the test set and is the database of all user “no-rating” item vectors; each vector is the set of items for which the user has yet to rate and for which we would like *RecTree* to produce predictions.

Output: A mapping of each element of each vector of *S* into a rating score or “no rating”, *Θ*.

Method: A prediction is computed in the following manner:

```
For each user $\boldsymbol{r}_a$ in the test set, $S$:
    For each item $i$, in $\boldsymbol{r}_a$:
        i.   Create a recommendation message, recMess(a, i, ?)
        ii.  Descend the RecTree to the maximum depth.
        iii. If the user is located in an internal node, $C_k$, and $c_k$ is the node centroid then
                 return $p_{a,i} = c_k(i)$.
        iv.  If the user is located in a leaf node Call ComputeDeviationFromMeanPrediction(..)
        v.   If the user is located in an outlier node then
                 ascend up the link to the parent node, $C_k$ with centroid $c_k$.; return $p_{a,i} = c_k(i)$
```

Example 12.

In this example we generate Basil and Sammy’s predictions for *Matrix* and *Titanic* using the *RecTree*. Each user’s data structure has a pointer linking the user to the *RecTree* node that he resides in. Both users reside in leaf nodes (see Figure 14) and so the subroutine *ComputeDeviationFromMean(..)* is called to generate predictions. The weights for the predictions are derived from the similarity matrix of each leaf.

		Prediction		Actual		MAE
		Matrix	Titanic	Matrix	Titanic	
Users	Sammy	2.7	3.5	3	4	0.4
	Basil	4	4.4	4	5	0.3

Figure 15: The RecTree’s collaborative filter is trained as nodes are attached to the tree.

□

Theorem 12. The *RecTree* generates predictions with complexity O(β).
Proof. An internal node provides a recommendation in constant time by consulting its centroid. A leaf node provides a prediction by weighting all of the advisors in the partition. Given that the average size of a partition is β, the complexity is O(β). □

3.4.4 Updating the *RecTree*

An active CF system is continually collecting new item-ratings on existing users and increasing its membership. In this section we discuss how the *RecTree* is updated when a new user with a history of item-ratings is added to the system. An existing user who submits additional ratings or makes changes to his existing ratings can be modelled conceptually by deleting the user and then re-inserting the user into the *RecTree*.

A new user is added by descending the tree until the leaf node is reached or it is not possible to proceed. At each internal node, the user item-rating vector is compared to the cluster centres of each child node. The branch with the closest cluster centre is descended. If both branches are equally distant from the user, then neither branch is taken. The user is associated with the terminating node and the cluster centres on the path leading back up to the root are updated to reflect this new member.

If the user's descent reaches a leaf node V_L, a node splitting operation may be triggered if the node size exceeds the threshold β, otherwise the similarity matrix is simply updated with a call to *ComputeCorrelationSimilarity*$^+$*(..)*. The *ConstructRecTree(..)* routine is called to grow the sub-tree beneath V_L.

A user is removed from the *RecTree* by updating all the cluster centres on the path to the root. If the user resided in a leaf node, then the similarity matrix is updated by deleting the appropriate column and row entries.

3.4.5 Time Complexity

The *RecTree* can be constructed in O(n) and queried in constant time. Let $\underline{m}$ be the average number of item-ratings per user and l be the number of users in the dataset, then define n as the size of the data set given by $\underline{m}l$.

The partitioning and the collaborative training phases of our framework are interleaved in the construction of the *RecTree* data structure. By Theorem 10 and Theorem 11 the total complexity of these two phases is O(gtn) + O($n\beta$), which is linear with data set size. By Theorem 12 the *RecTree* can be queried in O(β), which is independent of the size of the data set n.

3.4.6 Space Complexity

RecTree has space complexity of O($g(m+1)$) + O(β^2k/m^2) where g is the maximum number of internal nodes, m is the number of items in the item set M, β is the partition size, l is the number of users, and k is the number of partitions.

Each branch of the tree requires constant space to store a link between a parent and a child node. From Lemma 3 and Corollary 5, we know that there are at most $2g$+1 nodes and therefore the storage requirements for the links is O(g).

Internal nodes require storage for their centroids, which at worst will be required to span the entire item space. If m is the number of items in the item set M then each centroid will require O(m) storage. There are at most g internal nodes and therefore O(gm) storage is required to store all of the internal nodes.

The leaf nodes require storage for the similarity coefficient matrix. This matrix as discussed in section 3.2.6 requires O(l^2/k) storage. Substituting for $\beta = ml/k$ the total space complexity is therefore:

$$O(g) + O(gm) + O(\beta^2k/m^2)$$

3.5 Chapter Summary

In this chapter, we introduced three new partitioned collaborative filters with the potential for linear complexity: *RandNeighCorr*, *KMeansCorr* and *RecTree*. Each of these algorithms partitions the data into *k* statistically independent partitions. Since only advisors within a single partition are consulted to make a recommendation, these algorithms have the potential to scale well. A detailed complexity analysis was presented and it was discovered that while all three algorithms have linear space complexity, only *RandNeighCorr* and *RecTree* have linear time complexity; the *KMeansCorr* algorithm is at best quadratic in time with data set size.

RandNeighCorr is a naïve partitioned collaborative filter. It is presented as the baseline algorithm against which the other partitioned collaborative filter will be measured against. A filter that does not surpass *RandNeighCorr* is not worthy of further consideration. *RandNeighCorr* randomly partitions the data set into *k* independent partitions. The correlation-based collaborative filter [RIS+94] is then applied within each partition to obtain recommendations.

KMeansCorr is a refinement of the RandNeighCorr algorithm. It replaces *RandNeighCorr*'s partitioning phase with the *KMeans*$^+$ clustering algorithm. It was discovered that the partitioning phase (consisting of the *KMeans*$^+$ algorithm), filter training phase (population of a correlation similarity matrix) and the prediction phase could not be simultaneously linearized. Linearizing one component made the other component at least quadratic in time.

RecTree is a new data structure that achieves linear time complexity by decoupling the parametric dependencies of the partitioning phase from the training and prediction phase. *RecTree* resembles an unbalanced binary tree with the leaves of the tree representing a partitioning of the data set and the path from the root to the leaves representing successive refinements in partitioning. *RecTree* employs a dual prediction strategy to improve its overall accuracy and to make it less susceptible to the *rating sparsity* problem.

Chapter 4 Results and Discussion

This chapter presents a performance study into the accuracy, coverage and running time of the new collaborative filtering algorithms, *RandNeighCorr* and *RecTree*. We compare each of these algorithms against the correlation-based collaborative filter, *CorrCF* [RIS+94]. In summary, we find that although *RandNeighCorr* is the fastest algorithm with linear scalability, its accuracy can be worse than non-personalized recommendations. *RecTree* in contrast outperforms *CorrCF* in accuracy, coverage and running time. In particular, *RecTree* exhibits linear scalability while *CorrCF* is quadratic with data set size.

This chapter is organized into 3 sections. In section 4.1 we discuss the methodology and the data set used for this study. In section 4.2 and 4.3 we present performance studies for *RandNeighCorr* and *RecTree*, respectively.

4.1 Methodology

We base our performance study on a comparison with the well-known correlation-based collaborative filter, *CorrCF* [RIS+94]. This filter has been demonstrated to be the most accurate of the nearest neighbour filters and has been incorporated into the GroupLens [KMM+97] and MovieLens recommendation systems [SKB+98].

In each of the performance studies that we present, a user's rating history is partitioned into two disjoint data sets; a test set consisting of 20 randomly selected item-ratings and a training set that comprises the remaining item-ratings. Once the collaborative filter has been trained, the test set is submitted to the filter for making recommendations. We compare the filter's predictions on the test set with the user's actual rating to obtain the accuracy and coverage. A discussion of the various accuracy metrics follows.

4.1.1 Accuracy and Coverage

The effectiveness of collaborative filters has traditionally been measured by their accuracy and their degree of coverage. Coverage is defined as the percentage of prediction requests that a filter can fulfill. Accuracy has been measured with a number of metrics that can be classified into statistical based and decision support based metrics.

A popular statistical accuracy metric is the mean absolute error (MAE) [RIS+94][SM95]. This measure is computed by averaging the absolute difference between the filter's recommendation and the user's actual vote. The mean square error (MSE) is a variation on MAE [RIS+94]; this measure is computed by averaging the square of the difference between the prediction and the actual vote. The rationale is that users will notice large deviations and their confidence in a system's utility will be greatly diminished by predictions that are significantly different from their actual rating. MSE penalizes a filter for large errors. The correlation between the recommendations and the actual ratings has also been used as an accuracy metric [RIS+94] [SM95]. The correlation measures the degree to which the filter's predictions track the user's actual ratings.

Reversal rate, F-number, and ROC sensitivity are examples of decision-support accuracy metrics. Reversal rate is the percentage of times the recommendations are very contrary to the user's actual rating. On a 5 point scale it could be defined as the percentage of predictions that deviate more than 3 points from the actual rating [SKB+98]. F-number is an information retrieval measure of accuracy that combines precision and recall, given by F = 2·precision·recall/(precision+recall) [SMc83]. Precision is the percentage of documents that are retrieved that are relevant and recall is the percentage of all relevant documents that are retrieved. In one CF application with an ascending 5 point rating scale, items with a score exceeding 4 were classified as relevant [BP99]. A F-number of 1 indicates perfect accuracy; every item that the user rated as relevant was correctly classified and only items that were relevant were presented to the user. ROC sensitivity is a signal processing measure first used by [SKB+98] to measure the decision support accuracy of the MovieLens movie recommendation system. The area under the sensitivity vs. 1-specificity curve gives the ROC sensitivity [Swe98]. Sensitivity is the probability that a randomly selected relevant item will be categorized as a relevant item by the filter. Specificity is the probability that a randomly selected irrelevant item will be categorized as an irrelevant item by the filter. A ROC sensitivity of 0.5 indicates an indiscriminate filter that is no better than random predictions. A score of 1 indicates a perfect filter.

Despite the myriad of accuracy metrics [GSK+99] reports that comparisons of different algorithms tend to be consistent regardless of the metric chosen. Given this observation, we choose MAE as our accuracy metric. It is straightforward to implement and seems to enjoy the widest support in use [RIS+94] [SM95] [Mil96] [KMM+97] [BHK98] [GSK+99] [PHL00].

4.1.2 Execution Time

The focus of this work is on the development of faster collaborative filters. The computationally intensive needs of traditional collaborative filters have necessitated that the similarity matrix, or user models be computed off-line. The GroupLens Usenet recommendation system employs separate process pools to compute predictions and similarity weights [KMM+97]. The rating process pool continually monitors a ratings log for new entries and assigns a process to update the similarity matrix as additional items are rated or new users join the system. The prediction process pool waits for requests and assigns a prediction process to compute a prediction using the current state of the similarity matrix. The recommendation is most accurate if the prediction process waits for any pending rating processes as this will allow the prediction process to take into account the ratings that the user has recently submitted. On the other-hand, this strategy imposes severe performance bottlenecks and with on exception is not used: when a user first joins the CF system the similarity matrix does not contain any references to him, consequently a prediction process cannot compute any predictions unless it waits for the similarity matrix to be updated by the rating process.

4.1.2.1 Batch-Mode

We present a hypothetical CF architecture whose running time is simpler to model than the GroupLens' architecture. In our hypothetical system, recommendations for all possible items are pre-computed. When a user requests an item-recommendation, a lookup into the recommendation table is performed in essentially constant time. If l is the number of users, and m is the number of items, the storage cost of such a strategy is O(lm). The EachMovie recommendation site accumulated nearly 73,000 users for 1600 movie titles over an 18-month period. Assuming that an item recommendation requires 6 bytes of storage (4 bytes to store the item id, and 2 bytes to store the numeric recommendation), pre-computing all of the item-recommendations would require approximately 700 megabytes of storage[6] (73000x1600x6) - well

[6] The storage requirements will in fact be significantly less since the sparsity of ratings will result in the "no-rating" prediction for a majority of items. The author's casual inspection of the sparsity suggests that a simple compression scheme such as run-length encoding could easily accomplish a compression factor of 10 to 1.

within the storage capability of currently available commodity hard drives. We refer to this mode of operation as batch mode. When a user rates additional items, the similarity matrix and recommendations are recomputed during the next batch window.

Batch mode is simulated in our study by clearing any data structures and asking the CF algorithm to repopulate using a new training set, *Y*. The CF algorithm then receives a test set, *S,* for which it computes predictions.

The batch mode execution time includes the time to train on the training set and the time to complete predictions on all of the test set. The test set consists of the same users as in the training set with 20 randomly withheld ratings per user. The batch mode running time for the training set of 200 users, for example, includes the time to train on 200 users and the time to complete 4000 predictions.

4.1.2.2 Interactive-Mode

When a new user joins the CF system we cannot wait until the next batch window to compute all potential recommendations. The CF system must respond immediately by locating his advisors and computing a prediction for the requested item. In contrast to batch mode, only a recommendation for the requested item is computed. We refer to this mode of operation as interactive mode.

Passing a single item-recommendation request for an as-yet unseen user to the CF algorithm simulates this interactive mode. The data structures are not cleared but updated as needed. The interactive mode assumes that the data structures to support recommendations have already been computed. This is the state of the system between batch processing windows. A new user joins the CF service, initializes his profile by rating some items, and then expects a recommendation. Typically, online recommendation sites ask the user to provide 10 to 20 ratings[7,8] before providing any recommendations. Obviously, a user can expect more and better recommendations if she provides more ratings on joining a service. In this work we studied the execution time for providing recommendations to new users who were required to provide 10 ratings to initialize their profile on joining the system.

[7] QRate: a free on-line service for movie recommendations asks for 12 initial ratings. This service can be accessed at: http://www.qrate.com

[8] Jester: an on-line service for recommending jokes asks for 15 initial ratings. This service can be accessed at: http://shadow.ieor.berkeley.edu/humor/.

4.1.3 Data sets

The data for this study is drawn from the EachMovie database [EM97]. This database was compiled over a 18-month period from the online movie and video recommendation site www.eachmovie.com. More than 2.8 million numeric ratings on 1,628 movie titles were compiled from a membership of 72,916 user. Each movie is rated on an ascending scale of preference from 0 to 5. Users have the option of supplying their age, gender and the zip code of their residence. Over 60% of the membership submitted more than 10 item ratings while only 10% submitted all three pieces of demographic data. We create a working set that consists of users each of whom have rated at least 100 items. We eliminated one user from the working set who consistently rated every item with a score of 4.

4.1.3.1 The GivenXUser Data Set

The purpose of this work is to evaluate how data size affects algorithmic performance of the collaborative filters. There are two dimensions along which data size can be varied: the number of users and the number of item-ratings per user. Along the user dimension we drew at random 200, 400, 600, 800, 1000, and 1400 users from the working set. We call these the GivenXUser data sets where X takes on the number of users in that data set. The item ratings of each GivenXUser dataset are partitioned into a training set and a test set. The training set consists of 80 randomly drawn item-rating per user while the test set consists of 20 randomly drawn item-ratings. The Given400Users data set, for example, is divided into a training set of 400 users with 80 randomly selected ratings and a test set of the same 400 users with 20 randomly drawn item ratings.

	GivenXUser Data Set			
	Training Set Size		Test Set	
Data Set Name	Users	Ratings / User	Users	Ratings / User
G200U	200	80	200	20
G400U	400	80	400	20
G600U	600	80	600	20
G800U	800	80	800	20
G1000U	1000	80	1000	20
G1400U	1400	80	1400	20

Table 9: The GivenXUser data set.

4.1.3.2 The GivenXRating Data Set

Along the item-rating sparsity dimension, we drew at random 40, 60, 80 and 100 item-ratings for each of 1400 users. We call these the GivenXRatings data sets where X takes on the number of item-ratings per user. As in the GivenXUser data sets, the item-ratings of each user are partitioned into a training set and a test set; the test set consists of 20 randomly drawn item-ratings while the training set retains the remainder of item-ratings. The Given60Ratings data set, for example, is divided into a training set of 1400 users each with 40 item ratings and a test set of the same 1400 users with 20 item ratings.

	GivenXRating Data Set			
	Training Set Size		Test Set	
Data Set Name	Users	Ratings / User	Users	Ratings / User
G20R	1400	20	1400	20
G40R	1400	40	1400	20
G60R	1400	60	1400	20
G80R	1400	80	1400	20

Table 10: The GivenXRating data set.

4.1.3.3 The InteractiveXUser Data Set

RecTree's interactive mode running time was tested as a function of the number of users in the system. As per the discussion in 4.1.2 we want to test the algorithm's performance when a new user joins the system. We took the GivenXUsers data sets and randomly withdrew 20 users from each data set to form the InteractiveXUser data sets; for clarity we refer to the reduced data set as GivenXUser′. Each InteractiveXUser data set is then partitioned into an interactive training set consisting of 10 randomly drawn item-ratings and a test set consisting of 20 randomly drawn item ratings. For example, 20 users are withdrawn from the Given200User data set to form the Interactive20User data set; each user in the Interactive20User data set has 10 item-ratings to train on and 20 item-ratings for testing.

We initially train *CorrCF* and *RecTree* on the GivenXUser′ data set. This initial training simulates the state of the CF system after a batch processing window. Once this training is complete we submit the InteractiveXUser data set to each of the filters for interactive mode prediction.

Data Set Name	GivenXUser' Training Set Size: Users	GivenXUser' Training Set Size: Ratings / User
G200U	180	80
G400U	380	80
G600U	580	80
G800U	780	80
G1000U	980	80
G1400U	1380	80

Data Set Name	InteractiveXUser Data Set Training Set Size: Users	Training Set Size: Ratings / User	Test Set: Users	Test Set: Ratings / User
I200	20	10	20	20
I400	20	10	20	20
I600	20	10	20	20
I800	20	10	20	20
I1000	20	10	20	20
I1400	20	10	20	20

Table 11: The InteractiveXUser data set.

4.1.4 The Partition Size Parameter β

We show in 3.2.5 and 3.4.5 that *RandNeighCorr* and *RecTree*'s complexity can be linearized if the partition size is kept constant. β is given by $l\underline{m}/k$ where l is the number of users, $\underline{m}$ is the average number of ratings per user and k is the number of partitions. If the value of β is fixed, then each of these factors must be varied to maintain the constancy of β's value.

An increase in the number of users can be adjusted for by decreasing the average number of item-ratings $\underline{m}$ in a partition, or by increasing the number of partitions k. The latter strategy is obvious and requires no further explanation. The former method however, is not feasible. The item-ratings for each user cannot be divided into smaller units since the entire history is required for computing similarity. Consequently, in the experiments utilizing the GivenXUser data set we maintain β constancy by varying the number of partitions k.

An increase in the number of ratings per user can be adjusted for by decreasing the number of users in a partition or equivalently, by increasing the number of partitions. In the experiments utilizing the GivenXRating data set, we also maintain β constancy by varying the number of partitions k.

4.1.5 Hardware & Software

The experiments for this thesis were conducted on a laptop PC with a Pentium III 600 MHz processor and 128 MB of RAM. The laptop has a 7 Gig hard drive. The laptop is running Windows 2000 Advanced Server operating system and all the software is written in Java using Borland's Jbuilder 3.5 and the Sun Java JDK 1.2.

4.2 Performance Study of *RandNeighCorr*

In this section we present a study into the performance of *RandNeighCorr.* In summary, we find that for non-degenerate values of β, *RandNeighCorr* is a fast linear collaborative filter that outperforms *CorrCF'*s running time in batch and interactive mode; *CorrCF,* however, always has superior coverage and accuracy. The administrator of a CF system can choose an operational point at which the trade-off between accuracy and running time is acceptable for his application. The value of β is bounded above by the data set size, *n* – choosing values larger than *n* causes *RandNeighCorr* to degenerate to *CorrCF* in operation. β is bounded below – choosing a partition size that is too small results in recommendations that are worse than non-personalized recommendations.

RandNeighCorr's accuracy is dramatically affected by item-rating sparsity; this study found that on data sets with fewer than 20 item-ratings per user, its accuracy was worse than non-personalized recommendations via population averages. The stability of *RandNeighCorr*'s overall accuracy was also affected by a growing database of users. As users are added to the system, the accuracy of *RandNeighCorr*'s predictions decline. This phenomenon confirms our analysis of 3.2.7.

4.2.1 The Partition Size β

RandNeighCorr's running time can be linearized if we constrain the partition size ($\beta=\underline{m}l/k$) to a constant (see section 3.2.5). We vary the parameter β and report on its effect on the algorithm's performance. If the value of β is greater than or equal to the size of the data set, the partitioning phase of *RandNeighCorr* puts all the data into one partition and the algorithm's performance degenerates to that of the un-partitioned collaborative filter, *CorrCF*.

We compare *RandNeighCorr*'s performance against *CorrCF* and that of *PopAvg*. *PopAvg* is a non-personalized recommendation algorithm that recommends an item based upon the average rating for that item. This algorithm makes no attempt to account for similarities between users in computing a prediction and computes the same recommendation for every user. We intend to use *PopAvg* to highlight *RandNeighCorr*'s performance at low item-rating densities.

RandNeighCorr partitions the users into independent clusters by random assignment, consequently the set of candidate advisors that are available for the prediction phase changes between each run of this algorithm. In some runs, very similar users were serendipitously grouped into the same cluster resulting in good recommendations and in other runs, a poor partitioning results in poor recommendations. Therefore, for each value of β we ran *RandNeighCorr* 10 times and reported the average performance measures.

4.2.2 *RandNeighCorr*'s Running Time

Figure 16 and Table 12 show the running time of *RandNeighCorr* in batch mode as a function of the data set size; these experiments were run on the GivenXUser data sets. The experiment was repeated 10 times for each value of β to obtain the average running times. We can clearly see that *RandNeighCorr*'s running time has a linear dependency on the size of the data set and the partition size β; these results confirm the complexity calculations of section 3.2.5. For the degenerate value of β = 16000 item-ratings, we see from Table 12 that the running time of *RandNeighCorr* is approximately the same as *CorrCF* for the data set with 16000 item-ratings For non-degenerate values of β, *RandNeighCorr* has a running time that is 4 to 36 times faster than *CorrCF*.

CorrCF has the slowest running time and is shown in Figure 17 with a quadratic fit. The running time for *CorrCF* is not shown in Figure 16, since it will skew the scale and hide the details for *RandNeighCorr*'s curves.

The running time for the non-personalized recommender *PopAvg*, is the fastest. This is to be expected since *PopAvg* requires only a single pass to compute all of the recommendations. *PopAvg*'s running time has a linear dependency on the data set size.

		Recommender				
				RandNeighCorr		
Data Set	Ratings	*CorrCF* *(sec)*	*PopAvg* *(sec)*	β=4000 (sec)	β=8000 (sec)	β=16000 (sec)
G200U	16000	211	4	78	114	207
G400U	32000	1213	11	157	221	478
G600U	48000	2515	18	233	365	872
G800U	64000	4235	25	288	499	1132
G1000U	80000	6321	32	322	645	1234
G1400U	112000	13991	45	389	718	1450

Table 12: *RandNeighCF*'s batch mode running time with number of item-ratings.

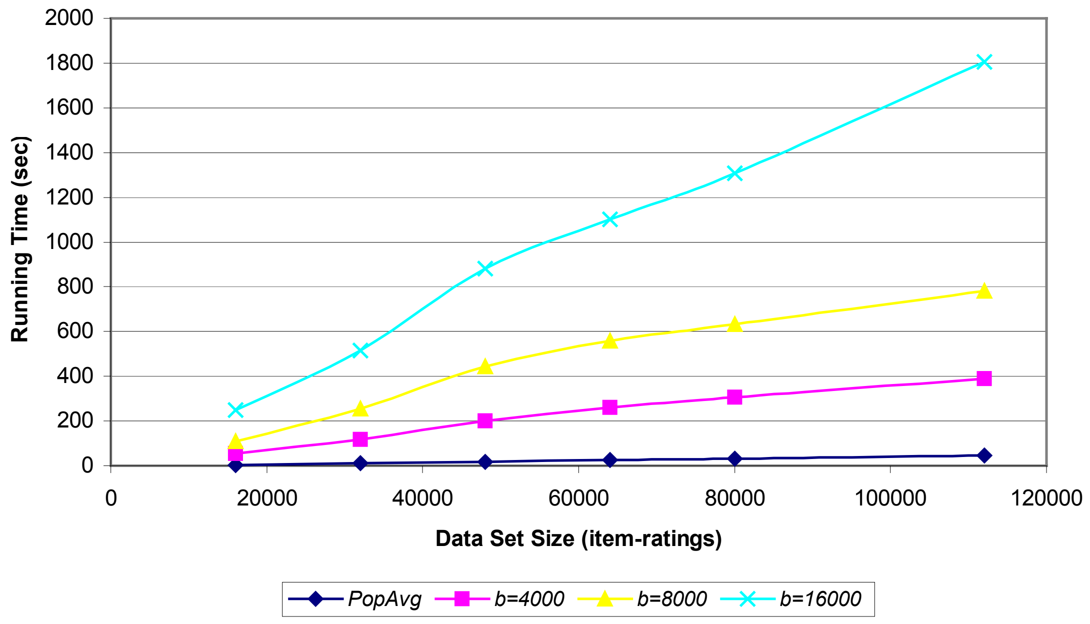


Figure 16: ***RandNeighCorr*****'s batch mode running time with number of item-ratings.**

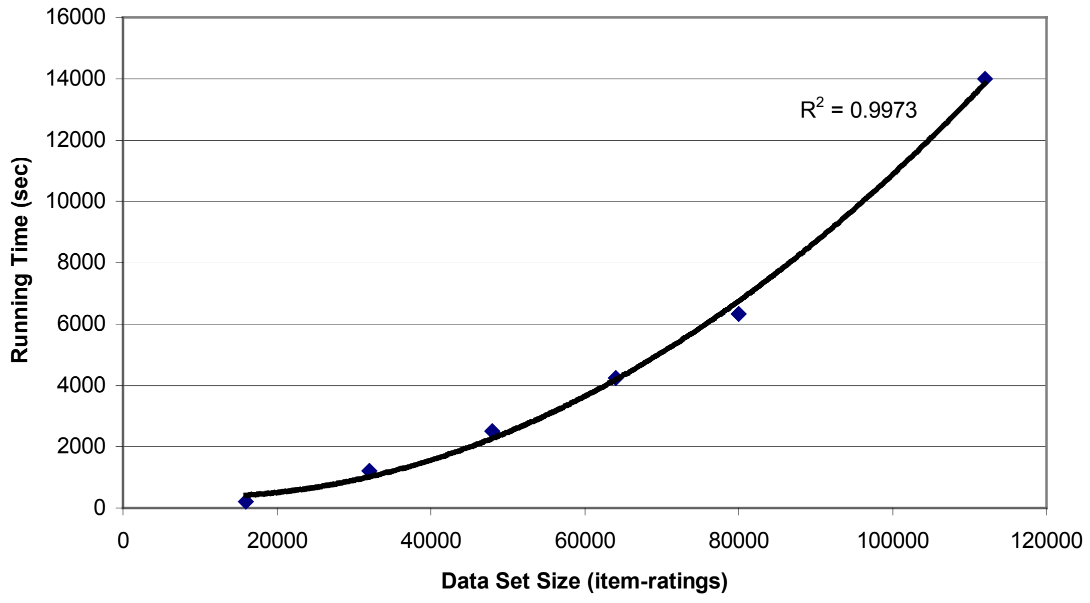


Figure 17: ***CorrCF*****'s batch mode running time is quadratic.**

4.2.3 *RandNeighCorr*'s Accuracy

We test the accuracy of *RandNeighCorr* by computing the MAE. Each experiment is repeated 10 times for each value of β and the MAE is then averaged. Smaller values of MAE indicate greater accuracy. In Figure 18, we show the accuracy of *RandNeighCorr* after training on the GivenXUser data set. The accuracy of *CorrCF* and *PopAvg* are also shown for comparison. Since the GivenXUser data set prescribes 80 item-ratings per user, we can equivalently express the partition size in terms of users. In Figure 18, the three separate runs of *RandNeighCorr* are shown with partitions of 50, 100, and 200 users.

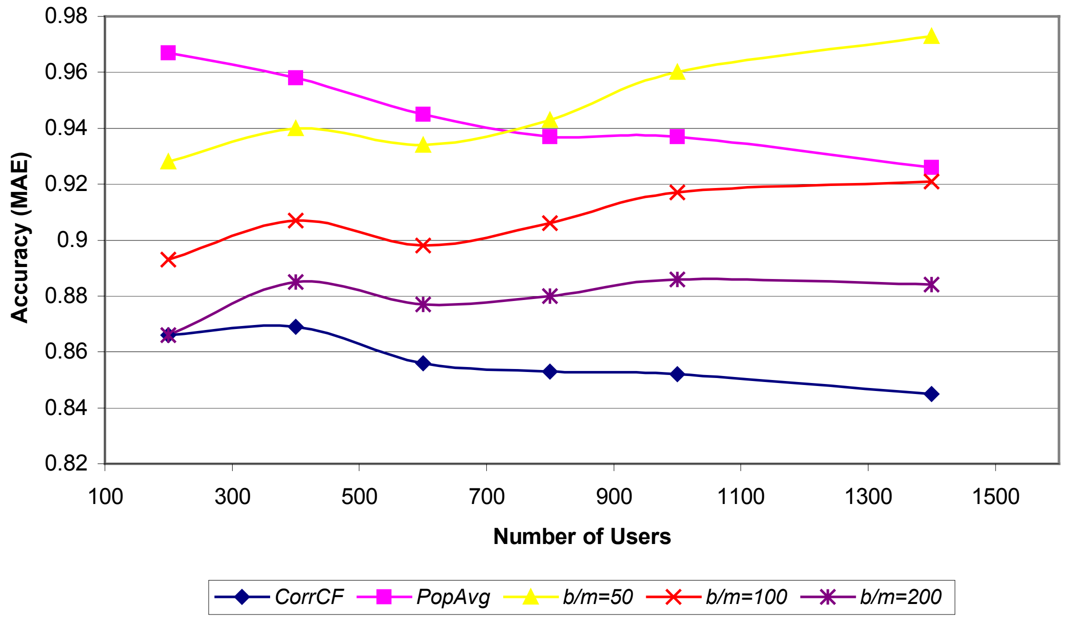


Figure 18: Accuracy of *RandNeighCorr* with number of users.

RandNeighCorr's accuracy improves as the partition size $\beta/\underline{m}$ increases. A larger partition allows the method *ComputeCorrelationSimilarity(..)* more candidates from which to find better advisors to make recommendations. As β increases to that of the data set size, the partitioning phase will put all of the data into a single partition and we expect *RandNeighCorr* to degenerate to collaborative filtering on un-partitioned data. This behaviour is demonstrated in Figure 18 for $\beta/\underline{m}$ = 200 users and at 200 users, where we see that *RandNeighCorr*'s accuracy matches that of *CorrCF*. For non-degenerate values of β, we note that *RandNeighCorr* is less accurate than *CorrCF* for all training set sizes. Each of *RandNeighCorr*'s curves exhibits an overall rising MAE (declining accuracy) with the number of users in the training set.

PopAvg's accuracy monotonically improves with the number of users in the data set. It is always less accurate than *CorrCF* over the range of this study. *PopAvg*'s accuracy is better than that of *RandNeighCorr* at $\beta/\underline{m}$ = 50 users and with a training set of 800 or more users. If we extrapolate *PopAvg*'s trend, it's accuracy will match or exceed *RandNeighCorr* at $\beta/\underline{m}$ = 100 users on a training data set with more than 1600 users.

RandNeighCorr's naïve approach to partitioning results in poor accuracy. By creating more partitions to accommodate larger data sets, the probability of locating the best advisors for each prediction is diminished – with the consequent reduction in recommendation accuracy. This unfortunate behaviour means that as the data set gets large, the accuracy of *RandNeighCorr* will at some point become worse than that of even non-personalized recommendations via population averages – this phenomenon is observed for the data sets and partition sizes tested. This behaviour confirms our analysis of 3.2.7.

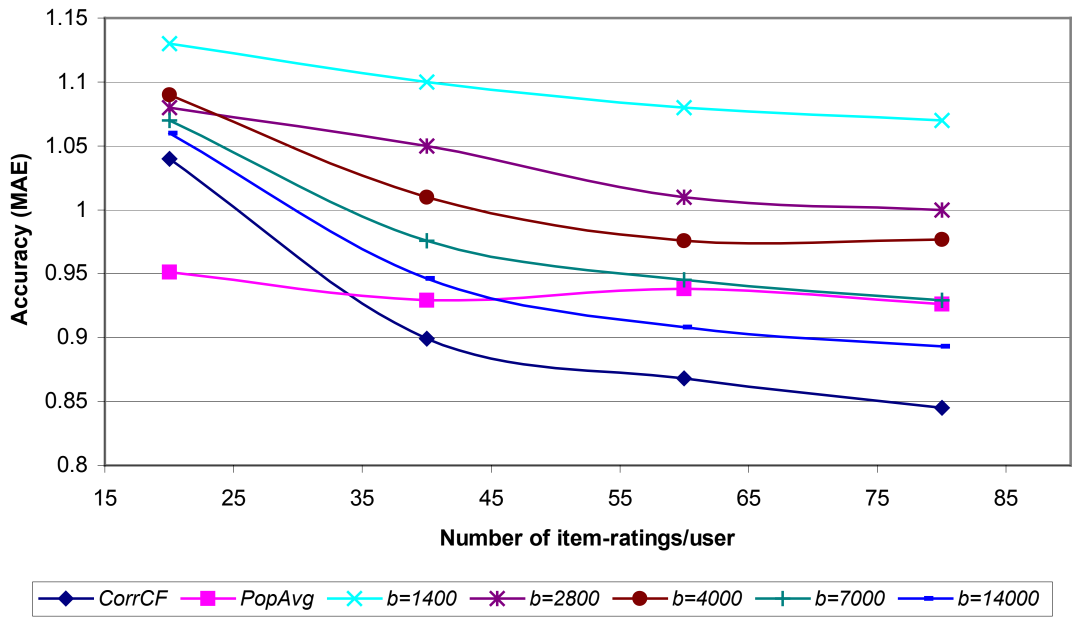


Figure 19: Accuracy of *RandNeighCorr* with number of item-ratings per user.

Figure 19 shows the accuracy with the number of item-ratings available per user (the GivenXItems data set). *RandNeighCorr* is executed on this data set with partition sizes fixed at

1400, 2800, 4000, 7000 and 14000 item ratings. The accuracy of *CorrCF* and *PopAvg* are also plotted for comparison.

The accuracy of *RandNeighCorr* and *CorrCF* monotonically improves with the number of item-ratings per user. More item-ratings allow for better recommendations since a more accurate calculation of similarity can be computed from longer item-rating histories. *RandNeighCorr*'s accuracy also improves with the size of the partition β, but its accuracy is always inferior to that of *CorrCF*. For this experiment, none of the values chosen for β are degenerate.

The non-personalized recommender, *PopAvg*, outperforms all of the personalized recommenders when the item-ratings are very sparse; *PopAvg* has the best accuracy at 20 item-ratings per user. *PopAvg* outperforms *RandNeighCorr* for the majority of the partition sizes tested and is only beaten by *RandNeighCorr* at β=14000 item ratings and at 40 or more item-ratings/user.

4.2.4 *RandNeighCorr*'s Coverage

The coverage of *RandNeighCorr, CorrCF,* and *PopAvg* is shown in Figure 20 as a function of the number of users in the data set. The coverage of *PopAvg* and *CorrCF* monotonically improve with more users. This occurs because more advisors are available to make recommendations. The coverage of *RandNeighCorr* is worse than *CorrCF* except for the degenerate value of β = 200 users and the data set with 200 users; Figure 20 shows that the coverage in this case is identical. *RandNeighCorr*'s coverage improves as the size of the partitions increases. A larger partition gives the collaborative filter more candidates from which advisors may be sought.

PopAvg's coverage is the greatest, since it can provide a recommendation on any item that at least one other person has rated. The personalized filters can provide recommendations only if an advisor has rated the item of interest.

Figure 21 shows the coverage as a function of the number of item-ratings per user. The coverage for all three algorithms monotonically improves with the number of item-ratings per user. The coverage increases since each advisor is able to provide more recommendations.

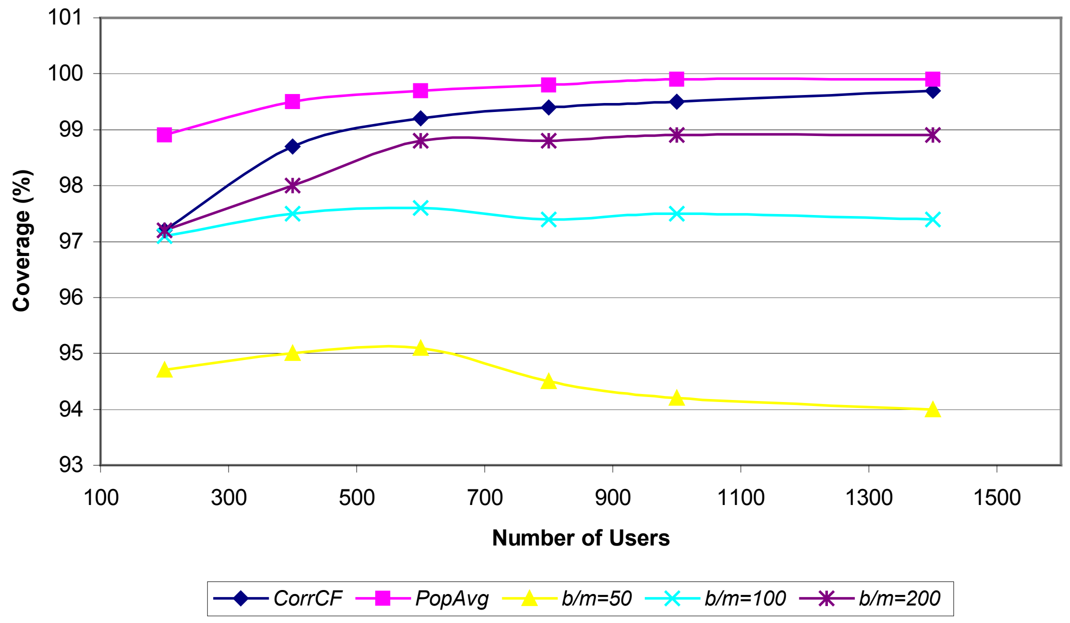


Figure 20: Coverage of RandNeighCorr with number of users. Each user has 80 item-ratings.

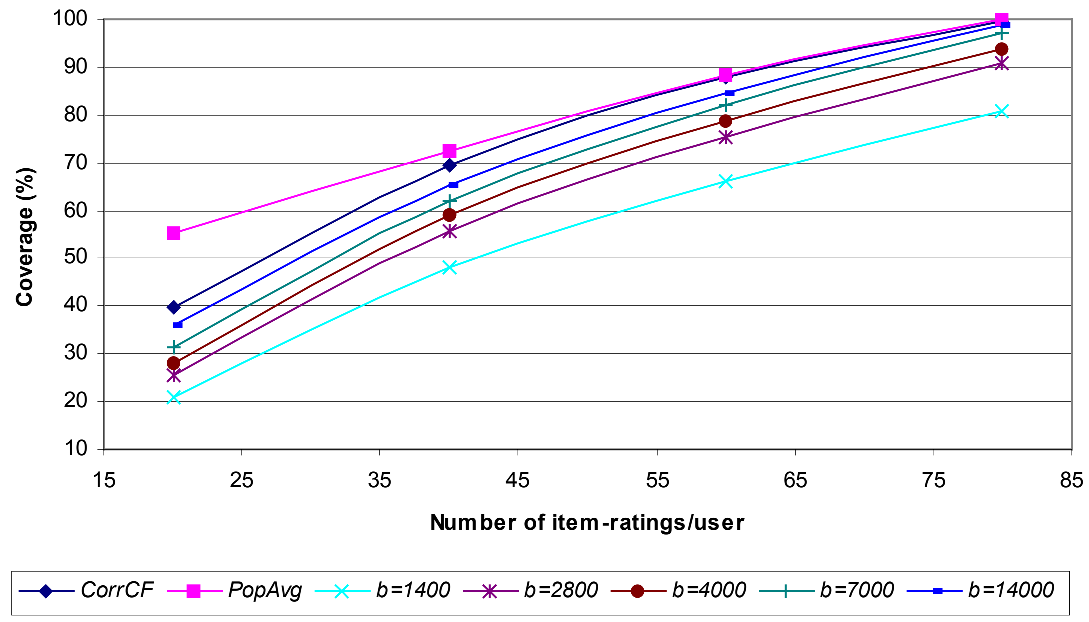


Figure 21: Coverage for *RandNeighCorr* with number of item-ratings per user.

4.2.5 Discussion

The accuracy of *RandNeighCorr* diminishes as users are added to the CF system. Our analysis of 3.2.7 shows that as the number of users increases, the probability of serendipitously putting highly correlated users in the same partition diminishes. We test this hypothesis further by plotting the average maximum similarity of advisors for each user as a function of partition and data set size in Figure 22. The plot provides more support for our hypothesis and indicates a declining average similarity as users are added to the system. The decline is sharpest for the smallest partition size.

RandNeighCorr makes a trade-off between running time and accuracy. We obtain predictions significantly faster in comparison to *CorrCF*, but the predictions are somewhat less accurate. Choosing an appropriate value for the partition size □ can set the operational point of the algorithm. For partition sizes greater than or equal to the data set size, *RandNeighCorr* degenerates in operation to *CorrCF*. For non-degenerate values of β, *RandNeighCorr* exhibits significant improvements over *CorrCF* in running time while sacrificing accuracy and coverage. However, if β is chosen too small in relation to the data set size, its accuracy falls below even that of non-personalized recommendations via *PopAvg*. The lower bound on β is determined empirically for each data set as it depends on the item-rating distribution.

PopAvg has the best accuracy, running time and coverage when the item-ratings density is low: *PopAvg* outperforms all of the personalized recommenders at less than 20 item-ratings per user.

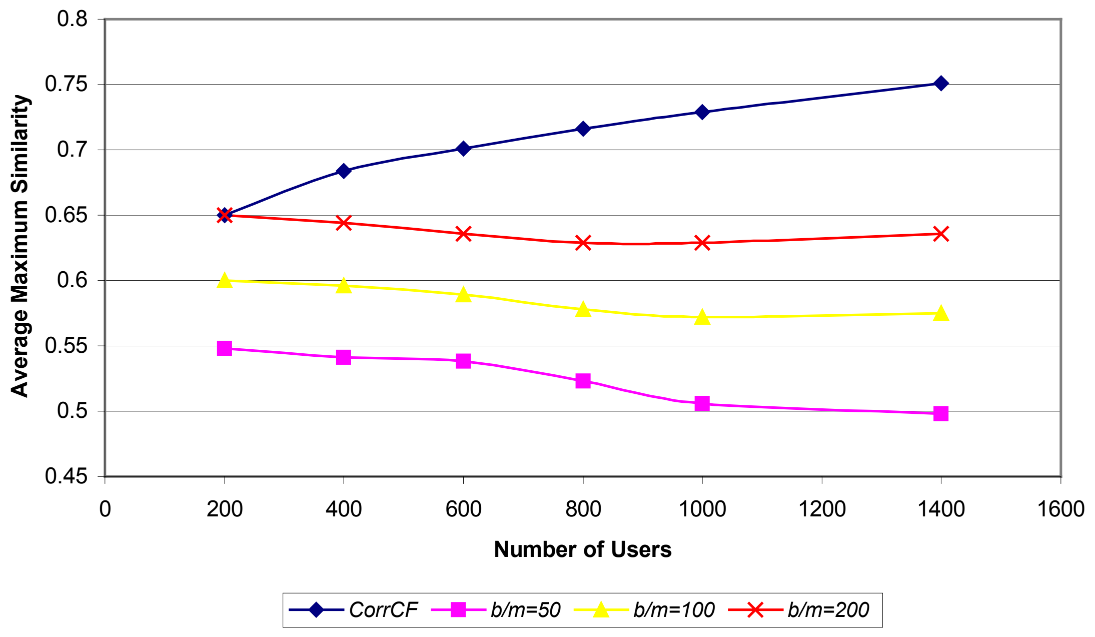


Figure 22: Average maximum similarity *RandNeighCorr* and *CorrCF*.

4.3 Performance Study of *RecTree*

In this section we present a performance study into *RecTree*. In summary, we find that *RecTree* significantly outperforms *CorrCF* in running time: *RecTree*'s linear complexity is confirmed. *RecTree* has accuracy and coverage that is superior to *CorrCF* regardless of the number of users and item-ratings. Furthermore, *RecTree* performs reasonably well even when item-rating density is low. *RecTree*'s performance can be tuned to trade-off accuracy for running time by adjusting the partition parameter β.

The construction of the *RecTree* requires the adjustment of three parameters: the partition size β, the internal node limit g and the outlier size limit *outlierSize*. In the following subsections we discuss how each of these parameters is set.

4.3.1 The Partition Parameter β

There are some values of β for which the *RecTree* is degenerate. If β is greater than or equal to the data set size, then the partitioning phase of *ConstructRecTree(..)* is bypassed and all of the data is placed into a single partition. We expect then that the accuracy, coverage and running time for degenerate values of β will be identical.

4.3.2 The Internal Node Limit *g*

The internal node limit *g* prevents *RecTree*'s construction algorithm from exhaustively partitioning pathological data distributions. Selecting a threshold that is too high may result in *RecTree* expending so much effort in partitioning the data that there will be no performance speed-up in comparison to *CorrCF*. Selecting a threshold that is too low may result in *RecTree* abandoning the partitioning phase pre-maturely. We use the following heuristic for selecting *g*.

If the data set has *n* item-ratings and the partition size has been set to β, then from Lemma 1 and Lemma 2 we have that the minimum and maximum number of partitions (leaf nodes) that *RecTree* can create is $\mathcal{L} = n/\beta$ and $\mathcal{H} = n-\beta+1$, respectively. We want to select a value of *g* between these two extremes such that *RecTree* will pursue a partitioning while it is profitable. We set *g* to the maximum of the following two values: $\mathcal{L}*s_1$ and $\mathcal{L}+(\mathcal{H}-\mathcal{L})*s_2$. For this study the values of $s_1={}^3/_2$ and $s_2={}^1/_3$ yields good values for *g*.

Example 13.
RecTree is given a training data set of 400 users, each user has 80 item-ratings to train on. The partition size has been set to 100 users. The minimum and maximum number of leaf nodes that can be created are $\mathcal{L}$=4 and $\mathcal{H}$=301, respectively. Our heuristic dictates that *g* will be set to the maximum of 6 (=4*${}^3/_2$) and 103 (=4+(297)*${}^1/_3$). *g* is therefore set to 103 iterations.

4.3.3 The Outlier Threshold: *outlierSize*

The *outlierSize* threshold prevents *RecTree* from computing predictions on small neighbourhoods. Predictions from small neighbourhoods tend to have poor coverage since there are relatively few potential advisors from which to draw recommendations and the accuracy of predictions tends to be low since the predictions are based on a small number of advisors

[HKB99]. Consequently, nodes with fewer members than *outlierSize* do not answer recommendation requests directly, but delegate such requests to their parent node. Section 3.4.3 discusses in detail how *RecTree* answers recommendation queries.

How do we choose an appropriate value for *outlierSize*? Figure 4 shows that for neighbourhoods with fewer than 20 users, the coverage declines sharply. [HKB99] shows that for the MovieLens data set, the prediction accuracy declined sharply with neighbourhood sizes smaller than 10 users. For this study we set the *outlierSize* threshold at 20 users.

4.3.4 *RecTree* Implementation

The sparsity of the item-rating space presents challenges to an efficient implementation of the *RecTree*. In this section we address the specific implementation choices that were made in this study.

4.3.4.1 The Item-Rating Vector

The user's item-ratings must be represented in a space and access-efficient data structure. An obvious choice for storing the item-ratings is an array. In this representation, each cell in the array stores either the user's rating for an item or the "no-rating" value Θ. The cell's ordinal position is the item identifier. The first cell in the array is special and stores the identifier of the user that "owns" this item-rating vector. In the item-rating vector below, User36 has rated item 1 with a score of 2 and item 3 with a score of 4. The remaining items have yet to be rated.

User36	2	Θ	4	Θ	..	…

Arrays have the advantage of being extremely fast to access since each cell in the array can be retrieved in constant time. The EachMovie database had a membership of 72,000 users and an inventory of 1628 movie titles. Storing the EachMovie ratings database in an array data structure would require ~ 200MB of storage. Although this storage requirement is easily met with commodity hard drives, it is highly inefficient; only 60% of the membership provided more than 10 movie ratings, resulting in an array of largely "no rating" value. In addition, it should be noted that the EachMovie recommendation site operated for only a brief period (approximately 18 months) and its membership is not large. A viable recommendation service will require significantly more memory to accommodate its operational need to both continually increase the membership and the inventory of items that users can rate.

Furthermore, it should be noted that the performance of the *RecTree* (and incidentally, *CorrCF*) degrades significantly if the database of item-rating vectors can not be loaded into memory in entirety[9]. Scanning the database of item-rating vectors from disk is at least an order of magnitude slower than reading that information from RAM. The array representation of the EachMovie database would fit into commodity RAM but would have limited capacity to grow with increasing membership and item inventory.

The shortcomings of the array approach and the high sparsity of the item-rating space motivated us to pursue a compressed data representation. In this approach, only the items for which the user has submitted ratings are represented in the data structure. A numeric pair consisting of the item identifier and the rating score represents each item rating. In the rating vector below, User45 has rated items 4, 7, 99, and 107 with the scores of 3, 2, 4, and 3 respectively. Items that the user has not rated are implicitly represented by their absence.

User45	4:3	7:2	99:4	107:3

This new representation is significantly more space efficient, but is not amenable to direct array access. One method of access is to use binary search on the item identifiers. This approach is relatively fast with a time complexity of O(log(N)), where N is the number of item identifiers. However, the array would need to be maintained in sorted order as users submit more ratings. We selected an alternative data structure, the binary tree. The binary tree also has a time complexity of O(log(N)) and the insertion and deletion of item-ratings is handled within the framework of the data structure. The Java JDK 1.2 provides an implementation of the binary tree and consequently reduced our development time.

4.3.4.2 The Similarity Matrix

RecTree answers a query in O(β) time. This complexity arises from having to scan through the similarity matrix for advisors. We improve on this search problem by representing the similarity matrix in a compressed format. A vector of numeric pairs represents each user's group of advisors; the first number is the advisor's identifier and the second number is the advisor's similarity score. In the similarity vector below, User88 has 3 advisor: User26, User28, and User98 with similarity scores of 0.67, 0.21, and 0.33 respectively.

[9] The assumption that the entire database of item-rating vectors will fit into memory is not necessariy unreasonable. Refer to 3.1.2 for detailed discussion.

User88	26:0.67	28:0.21	98:0.33

4.3.5 *RecTree*'s Batch Mode Running Time

The batch mode running time for *RecTree* is tested as a function of the number of users (the GivenXUsers data set) and of partition size. The experiment was run 10 times for each partition size and the average running time reported in Figure 23. Each of the users in the GivenXUser data set had 80 item-ratings, so we have equivalently expressed the partition parameter, β in terms of number of users. The batch mode running time for *CorrCF* is shown for comparison.

It can be clearly seen that *RecTree* outperforms *CorrCF* in running time for all values of $\beta/\underline{m}$ tested. The improvement is most dramatic for data sets with 1000 or more users. The significant improvement in speed is due to the linear complexity of *RecTree* in comparison to *CorrCF*'s quadratic complexity. A linear fit on each of the *RecTree* curves yields R^2 of 0.99 – empirically confirming the complexity analysis of 3.4.5. A quadratic fit on *CorrCF* curve yields R^2 better than 0.99.

The running time for *RecTree* improves with smaller partition sizes. For example, at 1400 users, the running time for $\beta/\underline{m}$ = 300 users is 5049 seconds in comparison to a running time of 4411 seconds when $\beta/\underline{m}$ = 100 users. In particular, the ratio of the partition size in relation to the size of the data set is an indication of the performance speedup that can be expected. The $\beta/\underline{m}$ = 200 curve at 200 users shows almost no improvement over *CorrCF*. However, at 1400 users, this same curve shows greater than 3 time reduction in the execution time in comparison to *CorrCF*.

The partition sizes $\beta/\underline{m} \geq 200$ users are degenerate. These partition sizes equal or exceed the size of the training set with 200 users and we note that the running time for *RecTree* is identical in these two cases.

The batch running time for *RecTree* and *CorrCF* as a function of item-ratings per user (the GiveXRatings data set) is shown in Figure 24. *RecTree* significantly outperforms *CorrCF'*s running time at all levels of item-ratings. *RecTree*'s linear dependency on item-ratings is confirmed with a linear fit. Each of the *RecTree* curves yield a fit of R^2 = 0.99. This linear dependency is in agreement with our analysis of 3.4.5. None of the partition sizes tested were degenerate.

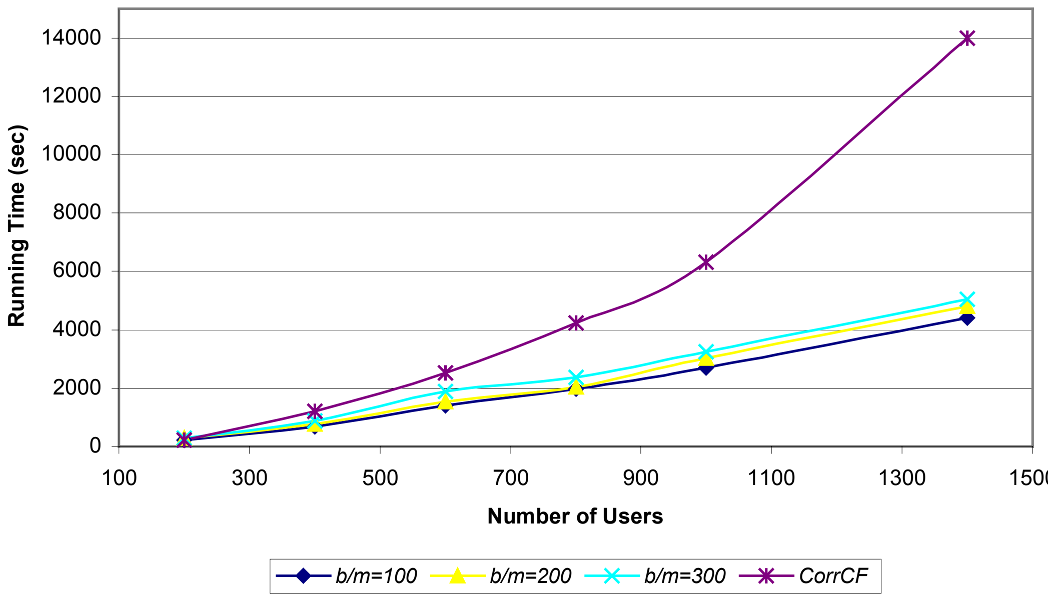


Figure 23: Batch mode running time for *RecTree* with number of users. *RecTree* demonstrates a linear complexity with number of users in contrast to *CorrCF*'s quadratic complexity.

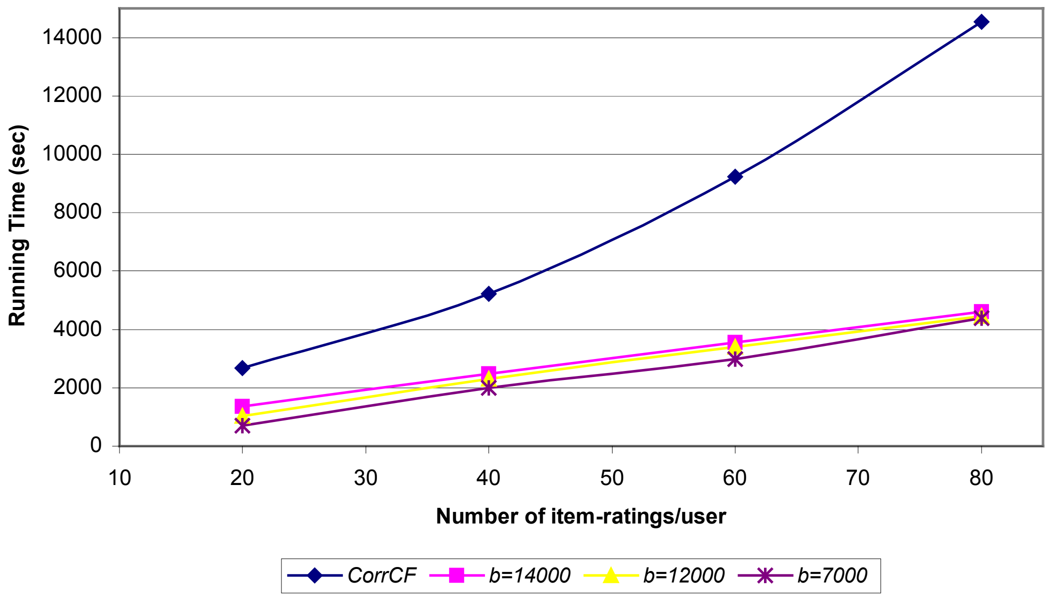


Figure 24: *RecTree* batch mode running time with number of item-ratings per user.

4.3.6 *RecTree*'s Interactive Mode Running Time

Figure 25 shows the running time for *RecTree* and *CorrCF* to compute 200 predictions in interactive mode as a function of the number of users already in the system.

RecTree significantly outperforms *CorrCF* in running time. Its response time is relatively constant with the number of users and shows only a dependency on the partition parameter β. From our analysis of 3.4.5 we know that querying the *RecTree* has a complexity of O(β). Our performance study confirms this analysis.

In comparison, the running time for *CorrCF* in interactive mode increases linearly with the number of users already in the system. For the largest data set, *CorrCF* is from 10 to 30 times slower than *RecTree*.

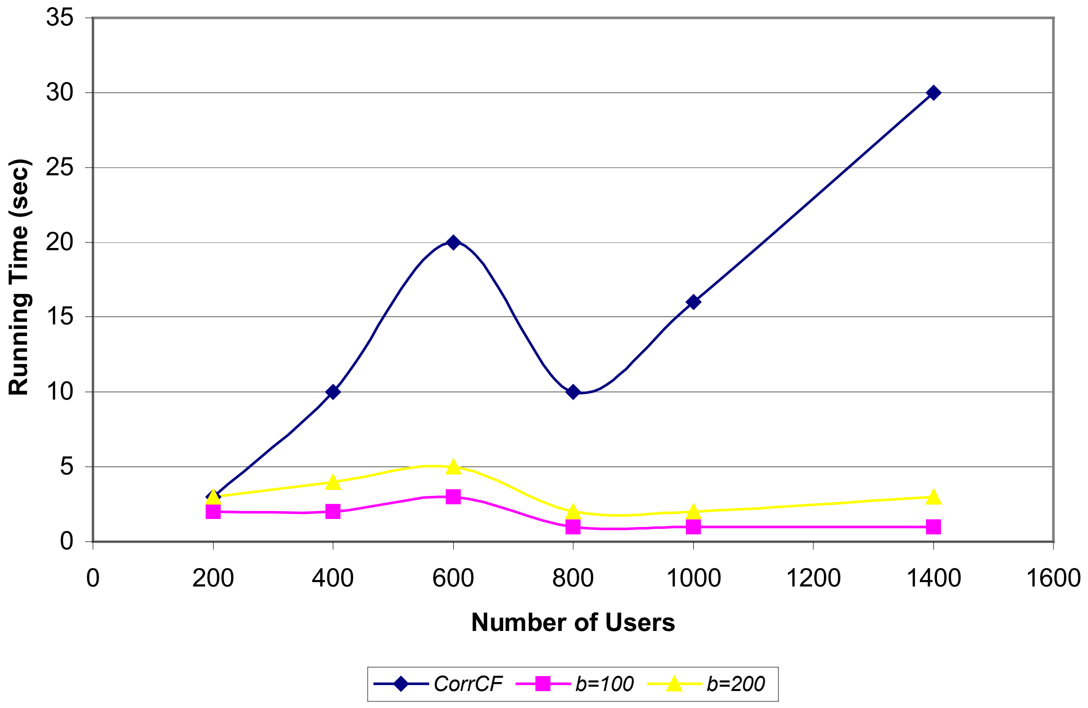


Figure 25: *RecTree*'s interactive mode running time

4.3.7 *RecTree*'s Accuracy

The accuracy of *RecTree* as a function of the number of users is shown in Figure 26. *RecTree* significantly outperforms *CorrCF* for almost all data set sizes and partition parameters $\beta/\underline{m}$. *RecTree*'s accuracy improves with larger partition sizes. *RecdTree*'s intelligent partitioning eliminates the degradation in accuracy with population increases that was evident in *RandNeighCorr*. A larger partition allows the algorithm more candidates among whom to locate good advisors.

For partition parameter $\beta/\underline{m} \geq 200$ users, *RecTree*'s accuracy is degenerate and we note that these curves have identical accuracy on a training set of 200 users.

The improvement in accuracy is due in part to the success of the partitioning phase of *RecTree* in localizing highly correlated users in the same partition. Figure 27 shows that the average similarity of advisors for *RecTree* is always higher than that of *CorrCF*.

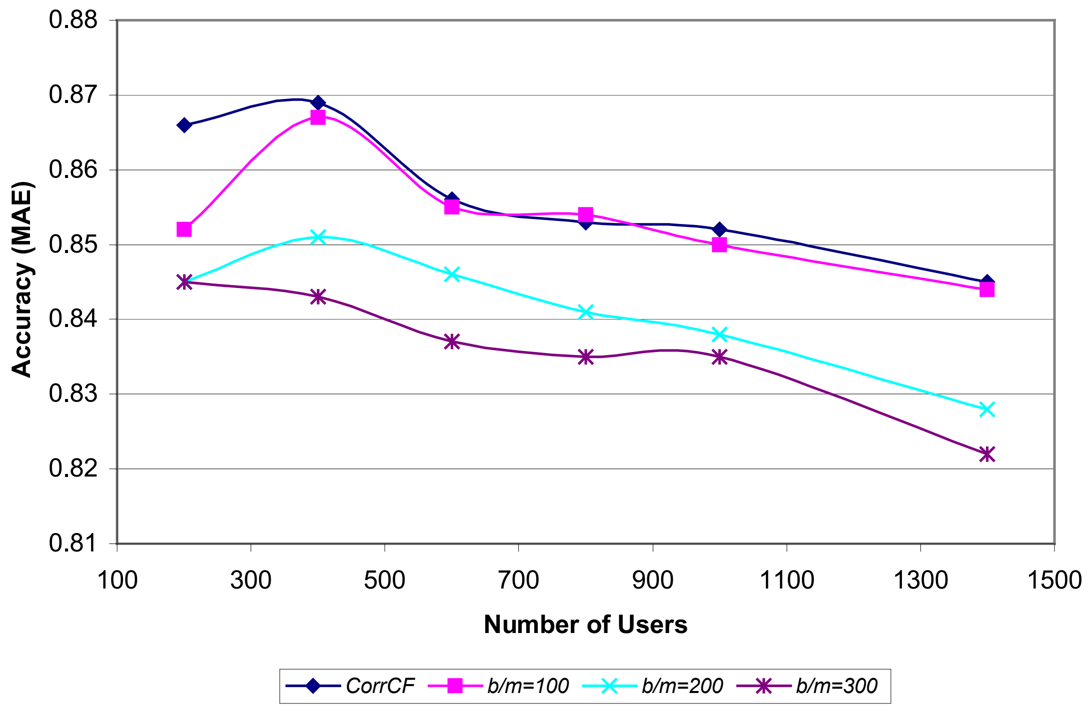


Figure 26: Accuracy of *RecTree* with number of users. β is in units of users.

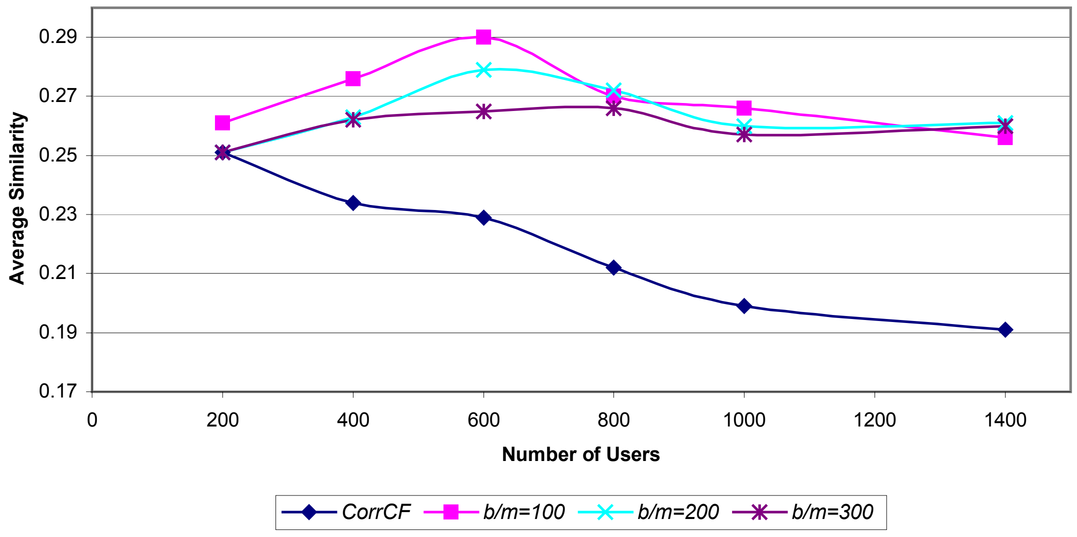


Figure 27: The average similarity of advisors in *RecTree*.

The accuracy of *RecTree* as a function of the number of item-ratings per user is tested with the GivenXRatings data set. This data set simulates the conditions of a CF system in the first months of operation: many users have joined the service, but each user has submitted only a few ratings. Traditionally, CF systems have poor accuracy during this period; [GSK+99] have referred to this situation as the *sparsity* problem. In 4.2.3 we show that for very sparse data, the recommendations for the personalized recommenders *CorrCF* and *RandNeighCorr* is worse than non-personalized recommendation via population averages, *PopAvg*.

The accuracy of *RecTree* as a function of rating sparsity is shown in Figure 28, with *CorrCF* and *PopAvg* also plotted for comparison. *RecTree* has very good accuracy even when the number of ratings per user is low. Its accuracy is always superior to *CorrCF* and *PopAvg* at all levels of rating sparsity.

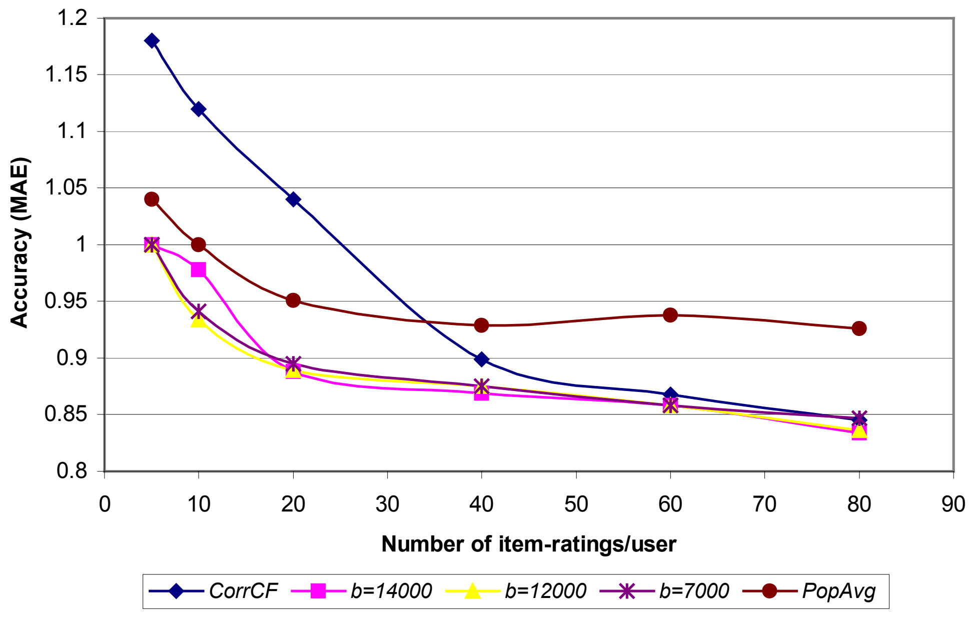


Figure 28: Accuracy of *RecTree* as a function of item-ratings per user.

4.3.8 *RecTree*'s Coverage

RecTree's coverage as a function of the number of users is shown in Figure 29. *RecTree* has very high coverage and is able to provide predictions more than 99% of the time for a majority of the data sets and partition sizes. At 200 users, *RecTree* outperforms CorrCF by almost 2% and for $\beta/\underline{m} \geq 200$, the coverage of *RecTree* is always greater than that of *CorrCF* for all data set sizes.

RecTree's coverage as a function of rating sparsity is shown in Figure 30. Its coverage is very good even when ratings sparsity is high; at 10 item-ratings per user, *RecTree* has a coverage that is almost double that of *CorrCF*. At 5 item-ratings per user, *CorrCF* has a slightly better coverage of roughly 11% in comparison to *RecTree*'s 9%.

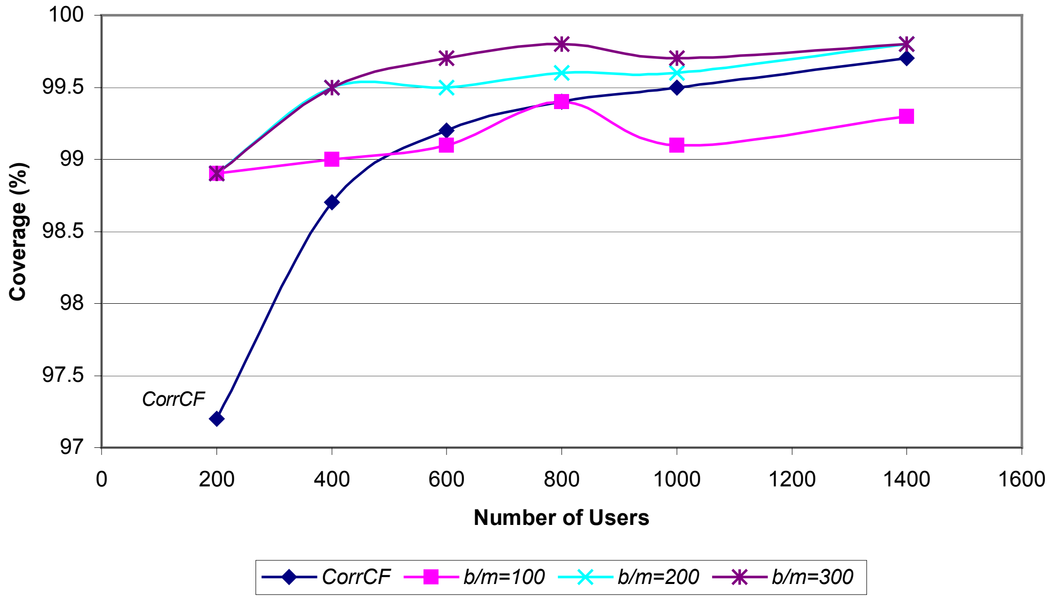


Figure 29: Coverage of *RecTree* with number of users.

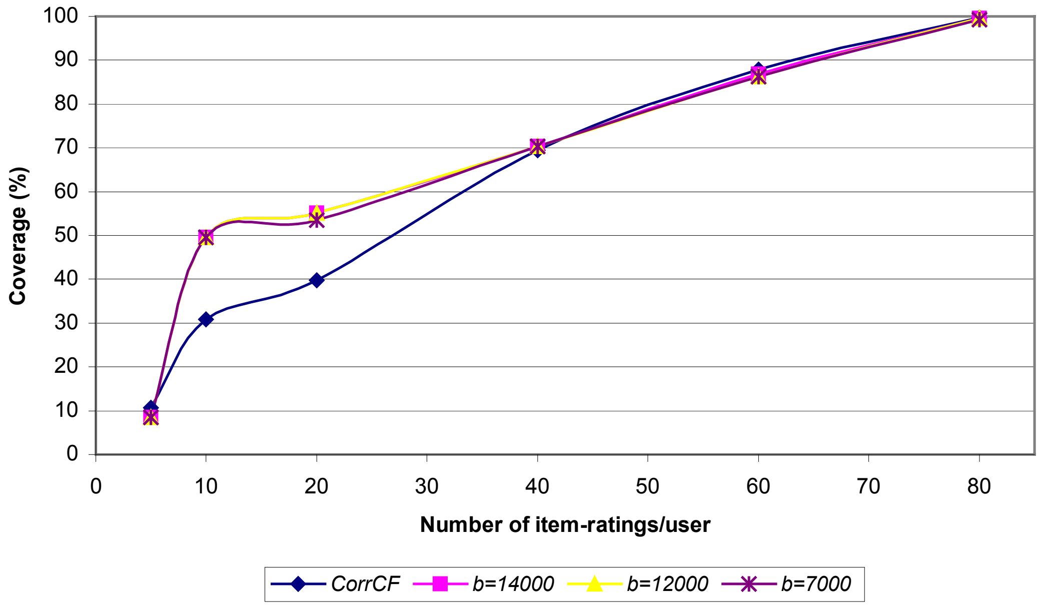


Figure 30: *RecTree*'s coverage with rating sparsity.

4.3.9 Discussion

RecTree can compute predictions in batch and interactive modes significantly faster than *CorrCF* and other nearest-neighbour collaborative filters. We show that in batch mode and for the largest data set, *RecTree* provides predictions in approximately 1/3 the time it takes *CorrCF*. *RecTree*'s linear scale up with number of users and item-ratings was confirmed. As each of these quantities increase, *RecTree* creates more partitions to accommodate the larger data set size.

In interactive mode, *RecTree*'s performance is even better: for the largest data set, predictions are provided in 1/10 to 1/30 of the time it takes *CorrCF*. *RecTree*'s constant query time with size of the data set was confirmed.

RecTree's superior running time arises from its partitioning of the user base into a set of independent clusters. Since advisors are sought only within a cluster, the time consuming task of computing similarity coefficients is substantially reduced. *RecTree*'s running time in batch mode is worse than interactive mode since it includes the time to grow the branches and the nodes of the tree. In interactive mode the existing tree is merely traversed for predictions. The time to construct the *RecTree* is dominated by the two processes of growing the tree branches and computing the similarity matrix at the leaf nodes. When the partition size is small the time required to compute the similarity matrix is reduced but more time is devoted to growing the tree. As we saw in Figure 23, Figure 24, and Figure 25, smaller partition sizes lead to improved execution time. Similarly, when the partition size is large *RecTree* spends less time constructing the tree branches but the time required to compute the similarity matrix increases; this strategy leads to increased execution times.

RecTree outperforms *CorrCF* in accuracy for almost all partition sizes and data sets. This achievement is the result of the three complimentary strategies of creating neighbourhoods of highly correlated users, computing pair-wise similarities based on each user's entire rating history, and avoiding spurious predictions from outlier nodes. *RecTree*'s partitioning phase is successful as the average similarity of advisors within a partition is higher than that of advisors in the un-partitioned case (c.f. Figure 27). A plot of the contributions of each of the latter two strategies to the overall accuracy of *RecTree* is shown in Figure 31.

RecTree has good accuracy even when there are few item-ratings. The accuracy of other collaborative filters diminishes greatly at low item-rating density, falling below even that of non-personalized recommendation via population averages [KM99a]. *RecTree,* on the other hand, maintains good accuracy even at low item-rating density. *RecTree* is able to deliver on this feature by avoiding spurious recommendations from outlier nodes. Figure 31 shows the accuracy with outlier node detection disabled.

RecTree outperforms *CorrCF* in coverage for almost all partition sizes and data sets. It achieves its high coverage due to its dual prediction strategy. *RecTree* initially seeks predictions from the user's advisors in the leaf node. If the user's advisors have not rated the item in question, then they cannot provide a recommendation. *RecTree* then delegates the recommendation request to the parent node where the larger group of users (to whom the user is similar to on a coarser granularity) may be able to satisfy the request. If the internal node cannot satisfy the recommendation request, the prediction is returned with a "no rating" value Θ. Obviously, if *RecTree* continued to pass the request up through the chain of parent-child links, it could increase its coverage even more. However, it was observed that the accuracy of the predictions degraded significantly when the prediction was delegated beyond the immediate parent node. Figure 32 shows the contribution of the delegation strategy to the overall coverage of *RecTree*.

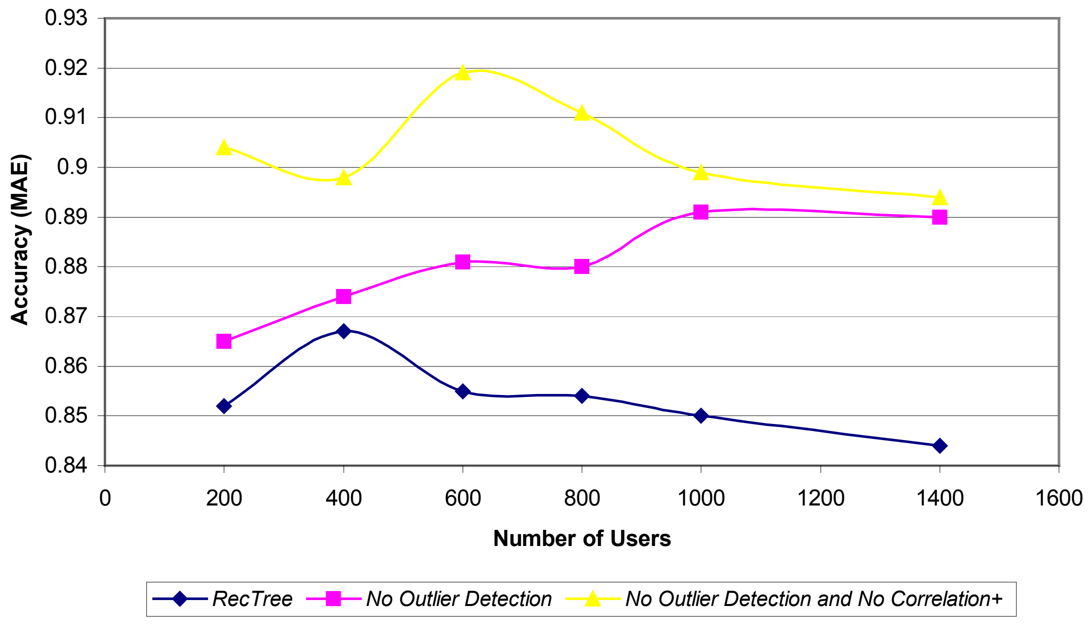


Figure 31: The contribution of correlation+ and outlier detection to *RecTree*'s accuracy.

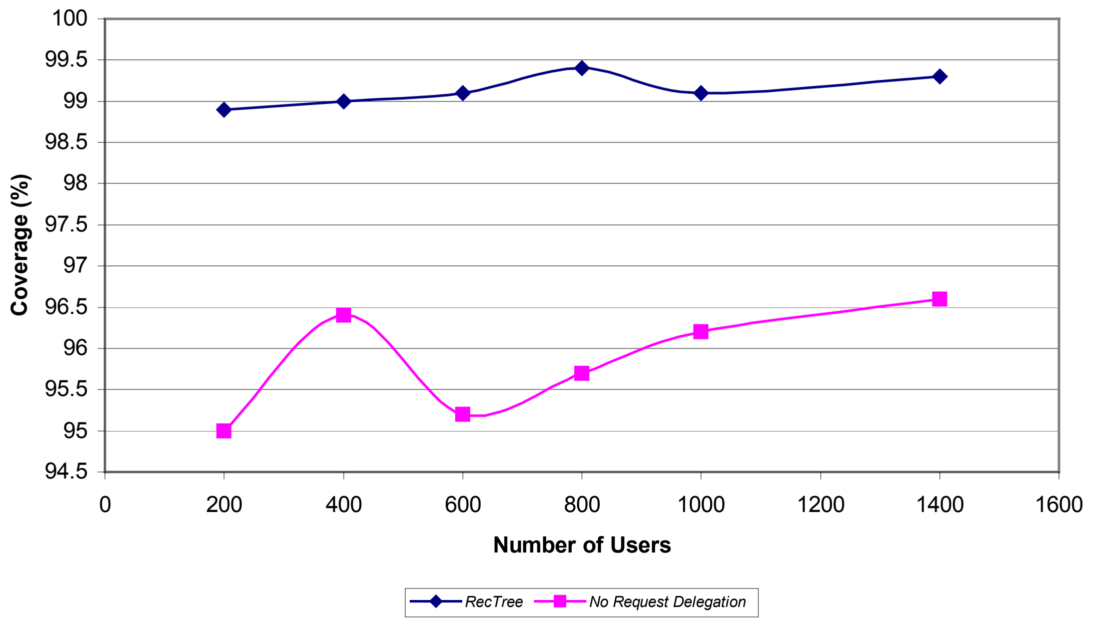


Figure 32: The contribution of *request delegation* to *RecTree*'s coverage.

4.4 Chapter Summary

In this chapter we presented a performance study of *RandNeighCorr* and *RecTree* and compared their performance against the baseline algorithm *CorrCF*. The linear time complexity of each of these algorithms with data set size was confirmed.

RandNeighCorr has the fastest execution time since it partitioned the data with a single pass over the data set. However, its naïve approach to partitioning also resulted in poor accuracy and inferior coverage in comparison to *CorrCF*. It was shown that as the data set size increases, *RandNeighCorr*'s accuracy (for a fixed partition size β) monotonically decreases. By creating more partitions to accommodate larger data sets, the probability of locating the best advisors for each prediction is diminished – with the consequent reduction in recommendation accuracy. This unfortunate behaviour means that as the data set gets large, the accuracy of *RandNeighCorr* will at some point become worse than that of even non-personalized recommendations via population averages – this phenomenon is observed for the data sets and partition sizes tested.

RecTree outperforms *CorrCF* in batch and interactive mode for almost all data sets and partition sizes tested. It not only has linear execution time scale-up, but also monotonic improvement in accuracy with data set size. *RecTree*'s superior accuracy arises from the success of the partitioning phase in creating neighbourhoods of highly correlated users, the use of longer rating histories for computing user similarity, and the avoidance of spurious recommendations from outlier nodes. *RecTree* has the unique capability among nearest-neighbour collaborative filters of providing accurate recommendations even when the rating density is low.

Chapter 5 Conclusion and Future Work

In this chapter we summarize our contributions and discuss the direction for future research to extend this study.

5.1 Conclusions

This thesis describes a new collaborative filtering data structure and algorithm called *RecTree* (an acronym for RECommendation Tree) that to the best of our knowledge, is the first nearest-neighbour collaborative filter that can provide recommendations in linear time. *RecTree* has the following characteristics:

1. *RecTree* can be constructed in linear time and space.
2. *RecTree* can be queried in constant time.
3. *RecTree* is more accurate than the leading nearest-neighbour collaborative filter, *CorrCF* [RIS+94].
4. *RecTree* has a greater coverage (provides more predictions) than *CorrCF*.
5. *RecTree* does not suffer the *rating sparsity* problem.

It was demonstrated that *RandNeighCorr*'s naïve approach to partitioning data yields clusters of un-correlated users. The resulting predictions were less accurate and the coverage was smaller than that of collaborative filtering on un-partitioned data via *CorrCF*. *RecTree* by comparison, outperforms *CorrCF* significantly in both accuracy and coverage. *RecTree* uses three strategies to achieve this result. Firstly, it partitions the user base into clusters of highly correlated users using the *KMeans*$^+$ algorithm. Secondly, it computes a more accurate measure of user similarity based on the entire rating history rather than just the intersection set between users.

Lastly, it avoids computing spurious predictions based on too few advisors by identifying outlier clusters.

5.2 Future Work

Due to time constraints, only a limited performance study into the effectiveness and efficiency of *RecTree* was possible. In this section, we indicate the work that we defer to the future for the extension and improvement of the *RecTree* algorithm.

5.2.1 Scalability

RecTree cannot handle large databases. In particular, if the database of item-ratings vectors can not fit into memory in entirety, the algorithm *ConstructRecTree(..)* becomes very inefficient since each clustering iteration of the *KMeans*$^+$ algorithm would require a complete scan of the database from disk. We hope to pursue the adaptation of recent scaleable clustering algorithms [ZRL96] [GRS98] [AGG+98] [ACW+99] to *RecTree* to deal with very large databases.

5.2.2 The Internal Node limit *g*

The internal node limit *g* is introduced to protect *RecTree* from computing useless partitions for pathological data distributions. However, a single threshold may not be flexible enough. For example, if the data was a superposition of an exponential and Gaussian distribution, a single threshold may cause *RecTree* to abandon the partitioning prematurely. We leave it to future work to investigate the use of multiple thresholds for improving *RecTree*'s ability to deal flexibly with these cases.

Alternatively, we believe that studying new and more effective means of detecting pathological distributions is an interesting and important direction for extension to the *RecTree*.

5.2.3 The *outlierSize* Threshold

The *outlierSize* threshold was set to 20 users. It would be an important improvement in *RecTree*'s applicability if future work discovered a more disciplined approach to setting this important parameter.

5.2.4 Similarity Measures

RecTree uses correlation as the similarity measure between users, however other similarity measures should be investigated. The probabilistic similarity measure of [PHL00] is an interesting alternative that we leave to future work.

5.2.5 Predictions

RecTree computes predictions by taking a weighted sum of deviations from a mean. This prediction has proven to be quite effective, but other methods should be tested. The weighted average score [SM95] and the most probable rating [PHL00] are two interesting alternatives that we did not pursue due to time constraints. Furthermore, previous collaborative filters have proposed only linear functions for generating predictions. It would be interesting to investigate the effectiveness of non-linear mapping functions.

While we have shown that RecTree offers improvements in accuracy, coverage and speed over its competitors, it is clear that there remains room to extend this lead even further.

□

Bibliography

[ACW+99] C. C. Aggarwal, C. Procopiuc, J. L. Wolf, P. S. Yu, and J. S. Park, Fast Algorithms for Projected Clustering, In *Proc. 1999 ACM SIGMOD Int. Conf. Management of Data (SIGMOD'99),* pages 61-72, Philadephia, PA, June 1999.

[AGG+98] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, Automatic Subspace Clustering in High Dimensional Data for Data Mining Applications, In *Proc. 1998 ACM SIGMOD Int. Conf. Management of Data (SIGMOD'98),* pages 94-105, Seattle, WA, June 1998.

[AZ97] C. Avery and R. Zeckhauser, Recommender Systems for Evaluating Computer Messages, *CACM*, 40(3), 88-89, March 1997.

[BC92] N. J. Belkin and W. B. Croft, Information Filteringa nd Information Retrieval: Two sides of the Same Coin?, *CACM*, 35(12), 29-38, December 1992.

[BF98] P. Bradley and U. Fayyad. Refining Initial Points for K-Means Clustering. http://www.research.microsoft.com/~fayyad/papers/icml98.htm. July 1998.

[BHK98] J. S. Breese, D. Heckerman, and C. Kadie, Empirical analysis of predictive algorithms for collaborative filtering. In *Proc. 14th Conf. Uncertainty in Artificial Intelligence (UAI-98)*, pages 43-52, San Franciso, CA, July 1998.

[BP99] D. Billsus and M. J. Pazzani, Learning collaborative information filters. *In Proc 15th Int. Conf. Machine Learning*, pages 46-54, Madison, WI, 1998.

[BS97] M. Balbanovic and Y. Shoham, Fab: Content-based, Collaborative Recommendation, *CACM*, 40(3), 66-72, March 1997.

[DDF+90] S. Deerwester, S. T. Dumais, G.W. Furnas, T. K. Landauer, and R. Harshman, Indexing by Latent Semantic Analysis, *J. Am. Soc. Inf. Sci.* 41, 391-407, (1990).

[EKS+96] M. Ester, H-P. Kriegel, J. Sander, and X. Xu, A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, In *Proc. 2nd Int. Conf. Knowledge Discovery and Data Mining (KDD'96)*, pages 226-231, Portland, OR, 1996.

[EM97] http://www.research.digital.com/SRC/eachmovie/

[FD92] P. W. Foltz and S. T. Dumais, Personalized Information Delivery: An Analysis of Information Filtering Methods, *CACM*, 35(12), 51-60, December 1992.

[FOR99] Forrester Research Inc., Forester Technographics Finds Online Customers Fearful of Privacy Violations, http://www.forrester.com/ER/Press/Release/0,1769,177,FF.html, October, 1999.

[GNO+92] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry, Using Collaborative Filtering to Weave an Information Tapestry, *CACM 35*(12), 61-70, December 1992.

[Gre99] B. Greenman, Liar Liar, http://www.zdnet.com/yil/content/mag/9903/liar.html, 1999.

[GRS98] S. Guha, R. Rastogi, and K. Shim, CURE: An Efficient Clustering Algorithm for Large Databases, In *Proc. 1998 ACM SIGMOD Int. Conf. Management of Data (SIGMOD'98),* pages 73-84, Seattle, WA, June 1998.

[GSK+99] N. Good, J. B. Schafer, J. A. Konstan, A. Borchers, B. Sarwar, J. Herlocker and J. Riedl, Combining Collaborative Filtering with Personal Agents for Better Recommendations, In *Proc. 1999 Conf. American Association of Artifical Intelligence (AAAI-99)*. July 1999 .

[HCC97] J. Han, J. Chiang, S. Chee, J. Chen, Q. Chen, S. Cheng, W. Gong, M. Kamber, K. Koperski, G. Liu, Y. Lu, N. Stefanovic, L. Winstone, B. Xia, O. R. Zaiane, S. Zhang, and H. Zhu, *DBMiner:* A System for Data Mining in Relational Databases and Data Warehouses, In *Proc. CASCON'97: Meeting of Minds*, Toronto, Canada, November 1997.

[HCC98] J. Han, S. Chee, and J. Y. Chiang, Issues for On-Line Analytical Mining of Data Warehouses, In *Proc. of 1998 SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'98)* , Seattle, Washington, June 1998, pages 2:1-2:5.

[HKB+99] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl, An Algorithmic Framework for Performing Collaborative Filtering, In *Proc. 1999 Conf. Research and Development in Information Retrieval*, pages 230-237, Berkeley, CA, August 1999.

[HM85] D. L. Harnett and J. L. Murphy, *Statistical Analysis for Business and Economics*, third edition, Addison Wesley, 1985.

[KM99a] A. Kohrs and B. Merialdo, Clustering for collaborative filtering applications. In *Computational Intelligence for Modelling, Control & Automation (CIMCA'99), Vienna*. IOS Press, 1999.

[KM99b] Arnd Kohs and Bernard Merialdo, Improving Collaborative Filtering with Multimedia Indexing Techniques to create User-Adapting Web Sites, In *Proc. 7th ACM Multimedia Conf., Orlando*. ACM, 1999.

[KMM+97] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl, Applying Collaborative Filtering to Usenet News, *CACM*, 40(3), 77-87, March 1997.

[KR89] L. Kaufman and P. Rousseeuw, *Finding Groups in Data, An Introduction to Clustering Analysis.* John Wiley and Sons, 1989.

[KSS97] H. Kautz, B. Selman, and M. Shah, Referral Web: Combining Social Networks and Collaborative Filtering, *CACM*, 40(3), 63-65, March 1997.

[MRK97] B. Miller, J. Riedl, and J. Konstan, Experiences with GroupLens: Making Usenet Useful Again. In *Proc. 1997 Usenix Technical Conf.*, January 1997.

[Paz99] M. Pazzani, A Framework for Collaborative, Content-Based and Demographic Filtering, *Artificial Intelligence Review*, 1999.

[PHL00] D. M. Pennock, E. Horvitz, S. Lawrence, and C. L. Giles, Collaobrative filtering by personality diagnosis: A hybrid memory and model-based approach, In *Proc. 16th Conf. Uncertainty in Artificial Intelligence (UAI-2000)*, Stanford, CA, June 2000.

[RIS+94] P. Resnick, N. Iacovou, M. Sushak, P. Bergstrom, and J. Riedl, GroupLens: An open architechure for collaborative filtering of netnews. In *Proc. ACM Conf. Computer Support Cooperative Work (CSC) 1994,* New York, NY, pages 175-186, October 1994.

[Rob81] S. E. Robertson, The methodology of information retrieval experiment. *Information Retrieval Experiment*, K. S. Jones, Ed., Chapter 1, pages 9-31. Butterworths, 1981.

[RP97] J. Rucker and M. J. Polanco, SiteSeer: Personalized Navigation for the Web, *CACM*, 40(3), 73-75, March 1997.

[RR91] R Rosenthal and R. Rosnow, *Essentials of Behavioral Research: Methods and Data and Analysis*, McGraw Hill, second edition, 1991.

[Sho92] L. Shoshana, Architechtng Personalized Delivery of Multimedia Information, *CACM*, 35(12), 39-50, December 1992.

[SKB+98] B. M. Sarwar, J. A. Konstan, A. Borchers, J. L. Herlocker, B. N. Miller, and J. Riedl, Using Filtering Agents to Improve Prediction Quality in the Grouplens Research Collaborative Filtering System. In *Proc. ACM Conf. Computer Support Cooperativ Work (CSCW) 1998*, Seattle, WA., page 345-354 Novemer 1998.

[SKR99] J. B. Schafer, J. Konstan, and J. Riedl, Recommender Systems in E-Commerce, *ACM Conf. Electronic Commerce (EC-99)*, Denver, CO, pages 158-166, November 1999.

[SMc83] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, 1983.

[SM95] U. Shardanand and P. Maes, Social information filtering: Algorithms for automating "word of mouth." In *Proc. 1995 ACM Conf. Human Factors in Computing Systems,* New York, NY, pages 210-217, 1995.

[Swe88] J. A. Swets, Measuring the accuracy of diagnostic systems, *Science*, 240(4857): 1285-1289, June 1988.

[THA+97] L. Tereen, W. Hill, B. Amento, D. McDonald, and J. Creter, PHOAKS: A System for Sharing Recommendations, *CACM*, 40(3), pages 59-62, March 1997.

[UF98] L. H. Ungar and D. P. Foster, Clustering Methods for Collaborative Filtering, In *AAAI Workshop on Recommendation Systems*, 1998.

[ZHL+98] O. R. Zaiane, J. Han, Z. N. Li, J. Y. Chiang, and S. Chee, MultiMedia-Miner: A System Prototype for MultiMedia DataMining, In *Proc. 1998 ACM-SIGMOD Conf. on Management of Data*, (system demo), Seattle, Washington, June 1998, pp. 581-583.

[ZRL96] T. Zhang, R. Ramakrishnan, and M. Livny, Birch: And Efficient Data Clustering Method for Very Large Databases, In *Proc. 1996 ACM SIGMOD Int. Conf. Management of Data*, pages 103-114, Montreal, Canada, June 1996.