RecTree: An Efficient Collaborative Filtering Method

Sonny Han Seng Chee, Jiawei Han, Ke Wang

Simon Fraser University, School of Computing Science, Burnaby, B.C., Canada
schee@cs.sfu.ca, han@cs.sfu.ca, wangk@cs.sfu.ca

Abstract. Many people rely on the recommendations of trusted friends to find restaurants or movies, which match their tastes. But, what if your friends have not sampled the item of interest? Collaborative filtering (CF) seeks to increase the effectiveness of this process by automating the derivation of a recommendation, often from a clique of advisors that we have no prior personal relationship with. CF is a promising tool for dealing with the information overload that we face in the networked world.

Prior works in CF have dealt with improving the accuracy of the predictions. However, it is still challenging to scale these methods to large databases. In this study, we develop an efficient collaborative filtering method, called *RecTree* (which stands for RECommendation Tree) that addresses the scalability problem with a divide-and-conquer approach. The method first performs an efficient k-means-like clustering to group data and creates neighborhood of similar users, and then performs subsequent clustering based on smaller, partitioned databases. Since the progressive partitioning reduces the search space dramatically, the search for an advisory clique will be faster than scanning the entire database of users. In addition, the partitions contain users that are more similar to each other than those in other partitions. This characteristic allows RecTree to avoid the dilution of opinions from good advisors by a multitude of poor advisors and thus yielding a higher overall accuracy.

Based on our experiments and performance study, *RecTree* outperforms the well-known collaborative filter, *CorrCF*, in both execution time and accuracy. In particular, *RecTree*'s execution time scales by $O(n\log_2(n))$ with the dataset size while *CorrCF* scales quadratically.

1 Introduction

In our daily life, virtually all of us have asked a trusted friend to recommend a movie or a restaurant. The underlying assumption is that our friend shares our taste, and if she recommends an item, we are likely to enjoy it. If a friend consistently provides good recommendations, she becomes more trusted, but if she provides poor recommendations, she becomes less trusted and eventually ceases to be an advisor. Collaborative filtering (CF) describes a variety of processes that automate the interactions of human advisors; a collaborative filter recommends items based upon the opinions of a clique of human advisors. Amazon.com and CDNow.com 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, etc. [13].

Example 1. A ratings database records a patron's reaction after viewing a video. Users collaborate to predict movie preference by computing the average rating for a movie from among their friends. A subset of the database is shown below where Sam and Baz have indicated a common set of friends. The average rating of *Matrix* is 3 while *Titanic* is 14/4. Therefore, *Titanic* would be recommended over *Matrix* to Sam and Baz.

		Titles				
		Speed (A)	Amour (R)	MI-2 (A)	Matrix (A)	Titanic (R)
Users	Sam	3	4	3		
	Bea	3	4	3	1	1
	Dan	3	4	3	3	4
	Mat	4	2	3	4	3
	Gar	4	3	4	4	4
	Baz	5	1	5		

Table 1: Higher scores indicate a higher level of enjoyment in this ratings database.

This simplistic approach falls well short of automating the human 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 the reliability of recommendations. Equally problematic is that the group average algorithm will make the same recommendation to all users. Baz, who has very different viewing tastes from Sam, as evidenced by his preference for action over romantic movies (as indicated by the letter A and R following each of the titles) will nevertheless be recommended *Titanic* over *Matrix*. Collaborative filters aim to overcome these shortcomings to provide recommendations that are personalized to each user and that can adapt to a user's changing tastes. □

Memory-based algorithms [3] are a large class of collaborative filters that take a list of item endorsements or a ratings history, as input for computation. These algorithms identify advisors from similarities between rating histories and then generate a recommendation on an as-yet unseen item by aggregating the advisors' rating. Memory-based collaborative filters differ in the manner that ratings are defined, the metric used to gauge similarity, and the weighting scheme to aggregate advisors' rating.

In the well-known correlation-based collaborative filter [11], that we call *CorrCF* for brevity, a 5-point ascending rating scale is used to record user reactions after reading Usenet items. Pair-wise similarity, $w_{u,a}$, between the user, *u,* and his potential advisor, *a*, is computed from Pearson correlation of their rating histories.

$$w_{u,a} = \sum_{i \in Y_{u,a}} \frac{(r_{u,i} - \bar{r}_u)(r_{a,i} - \bar{r}_a)}{\sigma_u \sigma_a \mid Y_{u,a} \mid} \qquad (1)$$

where $r_{u,i}$ and $r_{a,i}$ is the user and advisor rating for item *i* while $\bar{r}_u$ and $\bar{r}_a$ is the mean ratings of each user; σ_u and σ_a is the standard deviation of each user's rating history, and $Y_{u,a}$ is the set of items that both the user and his advisor have rated. A recommendation, $p_{u,j}$ is then generated by taking a weighted deviation from each advisor's mean rating.

$$p_{u,j} = \bar{r}_u + \frac{1}{\alpha} \sum_{i \in Y_{u,a}} (r_{a,i} - \bar{r}_a) \cdot w_{u,a} \quad (2)$$

where □ is a normalizing constant such that the absolute values of the correlation coefficients, and $w_{u,a}$ sum to 1.

The computation of the similarity coefficients can be viewed as an operation to fill in the entries of an *n* by *n* matrix where each cell stores the similarity coefficient between each user and his *n-1* potential advisors. Each row of the matrix requires a minimum of one database scan to compute and to fill the entire matrix of *n* rows therefore requires $O(n^2)$ operations. The computation of these similarity coefficients is the performance bottleneck in all previously published memory-based algorithms.

RecTree solves the scalability problem by using a divide-and-conquer approach. It dynamically creates a hierarchy of cliques of users who are approximately similar in their preferences. *RecTree* seeks advisors only from within the clique that the user belongs to and since the cliques are significantly smaller than the entire user database, *RecTree* scales better than other memory-based algorithms. In particular, creating more cliques as the dataset size increases allows *RecTree* to scale with the number of cliques rather than the number of users.

In addition, the partitions contain users that are more similar to each other than to users of other partitions. This characteristic allows *RecTree* to avoid the dilution of opinions from good advisors by a multitude of poor advisors – yielding a higher overall accuracy. The trick then is to create cohesive cliques in an economical manner.

This paper is organized as follows. Section 2 reviews related work. Section 3 details the *RecTree* algorithm. Section 4 describes the implementation of the *RecTree* algorithm and the experimental methodology. Section 5 compares *RecTree*'s performance against *CorrCF*. We conclude in Section 6 with a discussion of the strengths and weaknesses of our approach and the direction of future research.

2 Related Work

Two of the first automated collaborative filtering systems use Pearson correlation to identify similarities between users of Usenet[11] and music album aficionados[14]. In [14], the constrained Pearson correlation is introduced to account for the implicit positivity and negativity of a rating scale. Ringo also provides an innovative solution that inverts the basic CF approach; music albums are treated as 'participants' that can recommend users to other music album participants.

When the rating density is low, most CF systems have difficulty generating accurate recommendations [11] [5]. Unlike the problem of scalability, however, rating sparsity is an open issue that has received significant research attention. [12] and [5] attempt to ameliorate this issue by using bots and agents to artificially increase the rating density. Bots assign ratings based on criteria such as the number of spelling errors, the length of the Usenet message, the existence of included messages [12] or the genre of the movie title [5]. Agents are trained, using IF techniques, to mimic the rating distribution of each user. An agent regenerates its ratings as it becomes better trained which may force large portions of the similarity matrix to be updated [5]. In both of these works, the relevancy of the bots' and

agents' ratings to a particular user is decided by the CF system as it identifies potential advisors.

In their recent paper, Goldberg et al. [4] describe Eigentaste, which for certain domains does not suffer from the sparsity and scalability problems. They note that rating sparsity is introduced during the profiling stage when users are given the freedom to select the items they rate. In contrast, the Eigentaste algorithm forces participants to rate all items in a *gauge set*. The dimensionality of the resulting dense rating matrix is reduced using principal component analysis to the first two dimensions. All of the users are then projected onto this eigen-plane and a divisive clustering algorithm is applied to partition the users into neighbourhoods. When a new user joins the system their neighbourhood is located by projecting their responses to the gauge set onto the eigen-plane. A recommendation is generated by taking neighbourhood's average rating for an item. Eigentaste is a linear collaborative filter and requires $O(j^2n)$ time to compute its cluster structure. For small values of j, the size of the gauge set, Eigentaste can be very fast.

Eigentaste is however limited in that it requires the definition of a gauge set. In the Jester recommendation service, the gauge set consists of a set of jokes. After reading a joke, each user can immediately supply a rating. However, there are few domains where the items of interest can be consumed so quickly and evaluated.

It is worth noting that many e-commerce sites provide a simplified form of collaborative filtering that is based on the complementary technologies of data warehousing and on-line analytical processing (OLAP). Often-seen examples of OLAP style collaborative filtering are the factoids that attempt to cross-sell/up-sell products: *Item X has been downloaded Z times*. These rudimentary filters make the implicit assumption that all users are equally good advisors to the active user. A more sophisticated approach would be to mine patterns from the database and data warehouse [7] and to use these as the basis of a recommendation to the user.

3 The *RecTree* Algorithm

RecTree is the acronym for a new data structure and collaborative filtering algorithm called the RECommendation Tree. The *RecTree* algorithm partitions the data into cliques of approximately similar users by recursively splitting the dataset into child clusters. Splits are chosen such that the intra-partition similarity between users is maximized while the inter-partition similarity is minimized. This yields relatively small cohesive neighbourhoods that *RecTree* uses to restrict its search for advisors – which represent the bottleneck in memory-based algorithms. *RecTree* achieves its $O(n\log_2(n))$ scale-up by creating more partitions to accommodate larger datasets - essentially scaling by the number of partitions rather than the number of users.

Prediction accuracy deteriorates when a large number of lowly correlated users contribute to a prediction. Herlocker et al. [8] suggest that a multitude of poor advisors can dilute the influence of good advisors on computed recommendations. The high intra-partition similarity between users makes *RecTree* less susceptible to this dilution effect – yielding a higher overall accuracy.

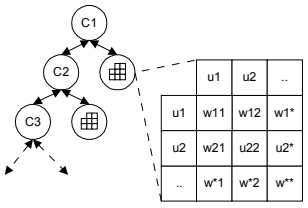

Fig. 1: The *RecTree* data structure.

The chain of intermediate clusters leading from the initial dataset to the final partitioning is maintained in the *RecTree* data structure, which resembles a binary tree. Within each leaf node, computing a similarity matrix between all members of that clique identifies advisors. *RecTree* then generates predictions by taking a weighted deviation from each clique's advisor ratings using (2).

Lemma 1. The training time for a partitioned memory-based collaborative filter is O(nb), where n is the dataset size if the partitions are approximately b in size and there are n/b partitions.

Proof. Training consists of computing a similarity matrix for each of the n/b partitions. Each partition is approximately b is size and computing the similarity matrix requires O(b^2) to compute. Computing all n/b similarity matrices is therefore O(nb). □

By fixing the partition size, Lemma 1 indicates that the training time for a memory-based collaborative filter can be linear in n.

3.1 Growing the *RecTree*

The *RecTree* is grown by recursively splitting the data set until a good partitioning of the data is obtained. It seems obvious that a clustering algorithm would be the ideal candidate for creating these partitions. Indeed, recent results have extended the applicability of clustering algorithms to high dimensions and very large disk-resident data sets [6] [2] [1]. However, given the sparsity of rating data and the low cost of RAM it is quite feasible to load all of the rating data into memory[1]; a fast in-memory clustering algorithm, such as KMeans [9] would therefore be appropriate for our needs.

KMeans begins its clustering by selecting k initial seeds as the temporary cluster centers and then assigning users to the cluster that they are closest to. The centroid of each cluster is then taken as the new temporary center and users are reassigned. These steps are repeated until the change in centroid positions fall below a threshold.

KMeans has a time complexity of O(k^2n) where k is the number of clusters and n is the dataset size. A naïve application of KMeans to create a proportionate number of cliques to match an increase in dataset size would yield cubic scale-up. Rather we employ KMeans as a procedure in a hierarchical clustering algorithm, which recursively splits the dataset into two child clusters as it constructs the *RecTree* from the root to its leaves. Since k is always 2 in this instance, KMeans is guaranteed to execute in time linear with dataset size. The procedure for constructing the *RecTree* is outlined in Algorithm 1.

Our purpose in selecting a clustering algorithm is neither to locate nor to identify the cluster structure in the dataset. We partition the data because we want to improve the execution time of the collaborative filter. The conditions for Lemma 1 guarantee linear training time but also restrict a clustering algorithm from obtaining optimal

[1] The EachMovie service over the course of 18 months accumulated over 70,000 users and an inventory of over 16000 movies, yet its entire rating database can be compressed and loaded into only 6 megabytes of RAM.

clusters. Despite this, our performance analyses show that *RecTree* is more accurate than un-partitioned collaborative filtering via *CorrCF*.

The *ConstructRecTree()* procedure is called recursively for each child cluster, *childClusterDataSet*, that is created by *KMeans()*. The tree continues to grow along a branch until the cluster is smaller than *partitionMaxSize*, or the branch depth, *curDepth*, exceeds the *maxDepth* threshold. The first condition ensures that all the cliques are approximately equal in size, which is essential to our efficient collaborative filter. The second condition ensures that *ConstructRecTree()* does not pursue a pathological partitioning.

Algorithm 1. *ConstructRecTree(parent, dataSet, partitionMaxSize, curDepth, maxDepth)*
Input: *parent* is the parent node from which the *dataSet* originates. *partitionMaxSize* is the maximum partition size and *curDepth* is the depth of the current branch. *maxDepth* is the maximum tree depth.
Output: The *RecTree*.
Method:

1. Create a *node* and link it to *parent*.
2. Assign *dataSet* to *node*.
3. If SizeOf(dataSet) ≤ *partitionMaxSize* OR *curDepth* > *maxDepth* then
   ComputeCorrelationMatrix(dataSet); RETURN.
4. *curDepth++*.
5. Call *KMeans (dataSet, numberofClusters=2)*
6. For each child cluster resulting from KMeans:
   Call *ConstructRecTree(node, childClusterDataSet, partitionMaxSize, curDepth, maxDepth)*.

Lemma 2. The *RecTree* data structure is constructed in $O(gn\log_2(n/b))$, if the maximum depth is $g\log_2(n)$. n is the dataset size, b is the maximum partition size and g is a constant.

Proof. We established in Lemma 1 that *RecTree*'s training phase (step 4) is linear. All that remains is to show the complexity of creating the partitions. At level one, the cost of creating two partitions is qn, where q is a constant and n is the dataset size. At each subsequent level of the tree, the complexity of building the branches is $qn_1 + qn_2 + .. qn_t$ where t is the number of partitions on a level. Since $n = n_1 + n_2 + \ldots n_t$, the cost of each subsequent level is also qn. For a balanced tree the maximum depth is $\log_2(n/b)$, which yields a complexity of $O(n\log_2(n/b))$. For an unbalanced tree, the maximum depth is n/b, which yields a complexity of $O(n^2/b)$ at worst. Since we constrain the maximum depth at $g\log_2(n/b)$, the total complexity is at worst $O(gn\log_2(n/b))$. □

3.2 Default Voting

In the collaborative filters proposed by [11] and [14], pair-wise similarity is computed only from items that both users have rated. If the item intersection set is small, then the coefficients are poor measures of similarity as they are based on few comparisons. Furthermore, an emphasis on intersection set similarity neglects the global rating behaviour that is reflected in a user's entire rating history. [8] accounts for small intersection sets by reducing the weighting of advisors who have fewer than 50 items in common. We approach this issue by extending each user's rating history with the clique's averages. This default voting effectively increases the similarity measure to cover the union of ratings rather than just the intersection set.

3.3 Generating a Recommendation

RecTree generates a prediction for the active user by locating his clique and then applying the weighted deviation from mean method as shown in (2).

Example 2. Table 2a shows the correlation between Sam and his friends. Without default voting, Bea and Dan are indistinguishable; they have voted identically on the three movies that Sam has seen. However, the standard deviation and average rating indicate that Bea is more volatile in voting than Dan or Sam. With default voting, the correlation is computed over the entire voting histories and captures the differences in global behaviour; Dan is assigned a higher similarity coefficient than Bea.

The ConstructRecTree algorithm partitions the users into the two groups consisting of (Sam, Bea, Dan) and (Mat, Gar, Baz). Table 2b shows the movie predictions that *RecTree* generates using default voting and this partitioning. *RecTree*'s predictions are in-line with our expectations and recommend the romantic title to Sam and the action title to Baz. In comparison to *CorrCF*, *RecTree* more clearly capture Sam's tastes by predicting a larger difference in preference for *Titanic* to *Matrix*. *RecTree* generates each of these predictions from considering only 2 advisors, while *CorrCF* considered 4 advisors. The reduced search space results in better execution time for *RecTree* and since the intra-partition similarity is high, the dilution of good advisors by a multitude of poor advisors is avoided.

		Titles							
		Speed	Amour	MI-2	Matrix	Titanic	Std.		Corr to
		(A)	(R)	(A)	(A)	(R)	Dev	Avg	Sam
Users	Sam	3	4	3	3	3.75	0.5	3.4	1
	Bea	3	4	3	1	1	1.3	2.4	0.21
	Dan	3	4	3	3	4	0.5	3.4	0.98
	Mat	4	2	3	4	3	0.8	3.2	-0.83
	Gar	4	3	4	4	4	0.4	3.8	-0.75
	Baz	5	1	5	3	3.75	1.7	3.6	-0.74

(2a)

	Titles	
	Matrix	Titanic
	(A)	®
Sam	2.72	3.55
Baz	4.15	3.73

(2b)

Figure 2a: Similarity coefficients computed with default voting capture global rating behavior. 2b) RecTree's movie recommendations.

4 Experiments and Performance Analysis

We base our performance study on a comparison with the well-known *CorrCF* algorithm. This filter has been demonstrated to be the most accurate of the memory-based filters [3] and has been incorporated into the GroupLens [10] and MovieLens recommendation systems [12].

The *maxDepth* parameter is constrained by 0 to be $gn\log_2(n/b)$. For these experiments, we found that setting g to a value of two protected the *RecTree* algorithm from pathological data distributions while creating partitions efficiently. The performance analysis shows that this choice not only yields better than quadratic scale-up, but also improves in accuracy over collaborative filtering on un-partitioned data.

4.1 The dataset

The data for this study is drawn from the EachMovie database (http://www.research.digital.com/SRC/eachmovie/), which consists of more than 2.8 million ratings on 1628 movie titles. We create a working set that consists of users who have rated at least 100 items. For these experiments we create a training set and test set by randomly selecting 80 and 20 ratings, respectively, from the rating history of each user in the working set.

4.2 The Off-line/On-line Execution Time

We present the performance of *RecTree* and *CorrCF* under the two regimes we call off-line and on-line operation. Off-line processing occurs when the systems require re-initialization. The systems train on the rating data and then compute predictions for all items that the user has yet to rate. When a user subsequently visits the systems, a lookup in constant time yields his recommendations.

In on-line processing the systems defer computing predictions; rather than exhaustively computing all recommendations, only a recommendation for the requested item is computed. This is quite often the case when a new user joins a recommendation service and a complete off-line processing may not be feasible.

The off-line and on-line execution time for *RecTree* as a function of the number of users and maximum partition size, b, is shown in **Fig. 3** and **Fig. 4**, respectively. The experiments were run 10 times for each partition size and the average running times reported. *RecTree* outperforms *CorrCF* for all partition and dataset sizes tested.

Like other memory-based collaborative filters, *CorrCF*'s quadratic off-line performance derives from its exhaustive search for advisors. In contrast, *RecTree* limits its search to within the partitions. As more users are added to the database, more partitions are crated to accommodate them. This strategy allows *RecTree* to scale by the number of clusters rather than the number of users.

RecTree's on-line performance is independent of the number of users already in the system. *RecTree* traverses its branches to locate the cluster closest to the new user and then takes a weighted aggregate of his advisors' rating to yield a recommendation. Since the cliques are approximately constant in size, the execution time is independent of dataset size. In contrast, *CorrCF* requires a scan of the entire database to aggregate all of the advisor ratings. As more users are added to the system, the cost of the scan increases.

RecTree's execution time improves with smaller partitions. Although the algorithm spends more time creating the cliques, this cost is more than offset by the savings in computing the similarity matrices. Smaller cliques however entail a reduction in the accuracy that we discuss in the next section.

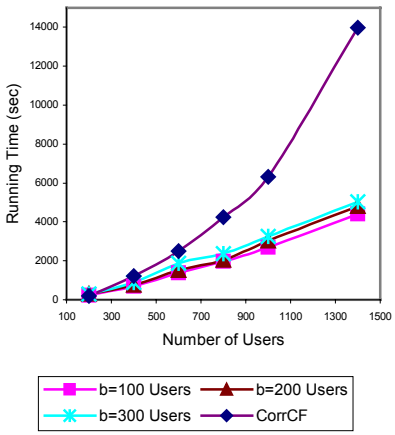

Fig. 3: Off-line performance of *RecTree*.

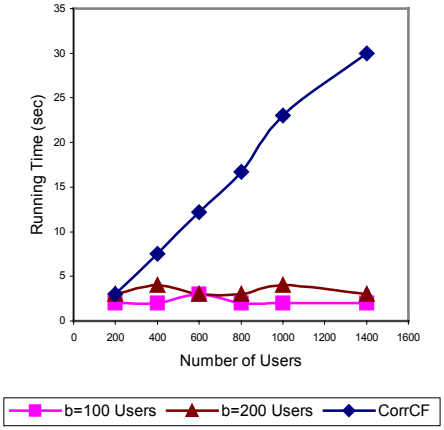

Fig. 4: On-line performance of *RecTree*.

4.3 The Accuracy Metric – NMAE

A popular statistical accuracy metric is the mean absolute error (MAE) [11] [14], which is the average absolute difference between the filter's recommendation and the user's actual vote. Goldberg et al. [4] proposes the normalized mean absolute error (NMAE), which normalizes the MAE by the rating scale. The NMAE has an intuitive explanation; it reflects the expected fractional deviation of predictions from actual ratings. The NMAE for a random filter applied to a random user, for example is 0.33 [4], which means that on average, we expect a prediction to be off by 33%. We use NMAE to report accuracy.

The accuracy of *RecTree* as a function of number of users and maximum partition size, *b*, is shown in **Fig. 5**; lower values of NMAE denote higher accuracy. *RecTree*'s improvement in accuracy with dataset size is typical of other collaborative filters. Larger datasets provide the CF algorithm with more candidates from which to select good advisors. The improvement however, is not necessarily monotonic; adding a batch of poorly correlated users will dilute the influence of existing good advisors. This dilution effect is clearly evident in the peak at 400 users; both *CorrCF* and *RecTree* for $b<300$ users show a drop in accuracy.

RecTree's accuracy is due in part to the success of the partitioning phase in localizing highly correlated users in the same partition. **Fig. 6** shows that the average similarity of advisors for *RecTree* is always higher than that of *CorrCF*. Because of the high intra-cluster similarity between users, *RecTree* is less susceptible to the dilution effect. At 1400 users in the dataset, *CorrCF* used an average of 223 advisors to compute a prediction. In contrast, *RecTree* for a partition size of 100 users, needed an average of only 46 highly correlated advisors.

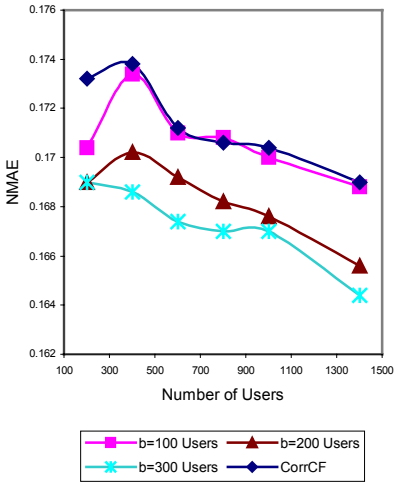

Fig. 5: NMAE of *RecTree* as a function of the number of users and partition size *b*.

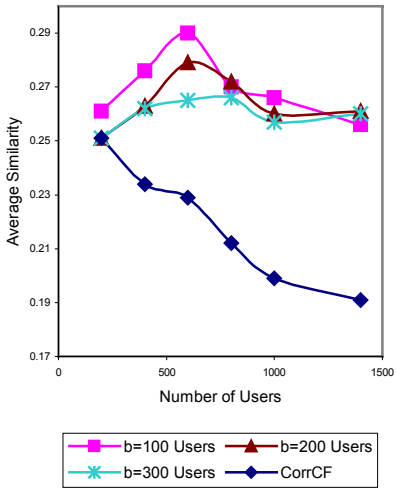

Fig. 6: The average similarity of advisors in *RecTree* exceeds *CorrCF*.

5 Discussion and Conclusion

In this paper, we describe, *RecTree*, a new linear CF method that applies clustering techniques to economically create cohesive cliques. *RecTree* achieves better scale-up in comparison to other memory based collaborative filters by seeking advisors only within a clique rather than the entire database. In particular, for off-line recommendation, *RecTree* scales by $O(n\log_2(n))$ and $O(b)$ for on-line recommendation, where n is the dataset size and b is the partition size, a constant.

RecTree achieves superior accuracy over *CorrCF* through default voting and the amelioration of the dilution effect. Default voting extends the rating history over which a similarity metric can be computed and hence improves the accuracy of a metric in capturing user proximity. The dilution effect is ameliorated by the high intra-partition similarity of a clique that acts as a coarse filter to limit the number of poor advisors that participate in computing a prediction.

Despite these improvements, we may yet be able to achieve higher accuracy by exploiting *RecTree*'s hierarchy of cliques. The current method computes predictions from only the members of the leaf cliques. We plan to investigate how the internal nodes of the *RecTree* data structure can contribute to even more accurate predictions.

The current implementation of *RecTree* assumes a single processor. However, the *RecTree* data structure can be easily distributed for parallel computation by a cluster of processors. For off-line processing, each branch of the tree can be grown independently of every other branch by assigning separate processing threads. For on-line processing, a thread can be assigned to each leaf node to handle prediction requests for new users. A parallel implementation of *RecTree* will be able to realize a greater throughput, which may be the subject of future work.

RecTree is an in-memory algorithm but for very large databases, it may not be an appropriate recommendation algorithm. It is the subject of future work to investigate the extension of *RecTree* to disk-resident databases with the incorporation of disk-based clustering algorithms, such as BIRCH [15].

References

[1] C. C. Aggarwal, C. Procopiuc, J. L. Wolf, P. S. Yu, and J. S. Park, Fast Algorithms for Projected Clustering, In *SIGMOD'99,* Philadephia, PA, June 1999.

[2] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, Automatic Subspace Clustering in High Dimensional Data for Data Mining Applications, In *SIGMOD'98,* Seattle, WA, June 1998.

[3] 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)*, pp. 43-52, San Franciso, CA, July 1998.

[4] K. Goldberg, T. Roeder, D. Gupta and C. Perkins, Eigentaste: A Constant Time Collaborative Filtering Algorithm, Information Retrieval, 2001.

[5] 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 *AAAI-99,* July 1999 .

[6] S. Guha, R. Rastogi, and K. Shim, CURE: An Efficient Clustering Algorithm for Large Databases, In *SIGMOD'9),* pp. 73-84, Seattle, WA, June 1998.

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

[8] 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*, pp. 230-237, Berkeley, CA, August 1999.

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

[10] 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, 1997.

[11] 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, Oct. 1994.

[12] 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., pp. 345-354 Nov. 1998.

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

[14] 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, pp. 210-217, 1995

[15] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: an efficient data clustering method for very large databases", In SIGMOD'96, Montreal, Canada, pp. 103-114, June 1996.