RecTree Centroid: An Accurate, Scalable Collaborative Recommender

Jerome Kelleher and **Derek Bridge**[1]

Abstract. We present a Collaborative Recommender that uses a user-based model to predict user ratings for specified items. The model comprises summary rating information derived from a hierarchical clustering of the users. We compare our algorithm with several others. We show that its accuracy is good and its coverage is maximal. We also show that the algorithm is very efficient: predictions can be made in time that grows independently of the number of ratings and items and only logarithmically in the number of users.

1 Introduction

Recommender Systems advise their users about which items (products, services or information) to consume. In *Content-Based Recommenders*, users articulate their requirements and the system matches them against item descriptions. In *Collaborative Recommenders*, which are the subject of this paper, no descriptions of requirements or items are needed; the system bases its recommendations on information about the preferences of related users. Standardly, if there are n users, $U = \{u : 1 \ldots n\}$, and m items, $I = \{i : 1 \ldots m\}$, preferences are represented using a $n \times m$ matrix r of ratings. User u may explicitly supply his/her rating of item i, $r_{u,i}$, or the system may obtain an implicit rating from observing user actions. A rating is typically represented by a Boolean or by a value from a numeric scale. Note that it is possible and common that $r_{u,i} = \perp$, signalling that the user has not yet rated that item. The user who is interacting with the recommender, $u_a \in U$, is called the *active user*. In the task of *prediction*, the recommender system uses r to compute a predicted rating $p_{u_a,i}$ for active user u_a and an item i for which $r_{u_a,i} = \perp$.

Collaborative Recommenders differ in the amount of work they do *off-line*, in advance of making predictions, and the amount of work they do *on-line*, when making predictions. Off-line, the system builds a *model* from r: the ratings data is mined for association rules, Bayesian networks or other structures that can capture regularities in the data. The model might be characterised as being either *user-based*, where the model captures relationships between users who have similar ratings, or *item-based*, where the model captures relationships between items that have been rated similarly. Then, on-line, predictions are made from the model.

In *model-based* systems, *most* of the work is done off-line; then, on-line, predictions are made using the model to the exclusion of the ratings matrix r. For example, Breese et al. [1] describe a system that learns a Bayesian network with a node for each item, arcs for item dependencies and decision trees at each node to encode the conditional probabilities. For on-line predictions, the items that the active user has rated are used as evidence variables, and the item-to-be-predicted, i, is the query variable.

In *memory-based* systems, little, if any, work is done off-line; predictions are made directly from r. An example of a purely memory-based system is the Exhaustive Recommender we describe in Section 3. It uses *neigbourhood-based* methods. Using the data in r, the active user's neighbourhood of recommender partners, i.e. the set of users who have rated i whose ratings are most highly correlated with the active user's ratings, is obtained. Then a prediction is computed from a weighted combination of the neighbours' ratings. Both of these computations happen on-line.

There are many ways of *combining* model-based and memory-based approaches. In Section 3, for example, we describe the possibility of an Exhaustive Recommender that pre-computes a matrix of all user-user correlations. This gives it a user-based model that speeds up the on-line process of finding neighbours. The approach still qualifies as memory-based because most prediction effort happens on-line and because, once the neighbours have been found, predictions are still computed directly from r.

In this paper, we propose a new approach that is purely model-based. A user-based model is built using clustering techniques; predictions are made using summary information about the clusters.

In Section 3, we describe a memory-based recommender. We show how it can be improved in Section 4. Section 5 explains the clustering algorithm we use in this work. Then in Section 6, we describe our new model-based algorithm, which uses summary information about the clusters for efficient, accurate predictions. Section 7 compares our results with those from related work. Before any of that, in Section 2 we explain how we have evaluated the different algorithms.

2 Evaluation

We compare the performances of the different algorithms that are described in this paper using two datasets: the MovieLens dataset (`www.grouplens.org`) and the PTV dataset[2]. Some of the characteristics of these datasets are reported in Table 1. The MovieLens

Table 1. Dataset Characteristics

Dataset	No. of users	No. of items	No. of ratings
MovieLens	943	1682	100000
PTV	2341	8164	60000

dataset has been cleaned up to include only users who have rated at

[1] Department of Computer Science, University College Cork. Email: `jt.kelleher|d.bridge@cs.ucc.ie`

[2] We are grateful to ChangingWorlds for supplying us with this dataset. The original dataset contains 60666 ratings but we removed 666 at random to give a round number of ratings.

least 20 movies; the PTV dataset has not. Both are sparse, but the PTV dataset is by far the sparser.

We have converted all ratings to z-scores in advance of using them in the experiments described in this paper. This conversion gives a very small improvement in the accuracy of the Exhaustive Recommender (Section 3) and significant improvements for the systems described in Sections 4 and 6.

In each experiment, the dataset is split into two disjoint sets, the training set (80%) and the test set (20%). All results are subject to five-fold cross validation, each time using a different 80/20 split. To model the on-going use of a recommender system, experiments use different total numbers of ratings. We report the effects on the accuracy, coverage and efficiency of the algorithms.

- To measure *prediction accuracy*, we use Mean Absolute Error (MAE). MAE is calculated by averaging the absolute difference between the algorithm's predicted rating and the user's actual rating (from the test set).
- *Prediction coverage* is measured by counting the number of times the algorithm fulfils a prediction request and reporting this as a percentage of the overall number of prediction requests made.
- To give an implementation-independent measure of *prediction efficiency*, we count Total Prediction Operations (TPO). We explain exactly which operations are counted in Sections 3 and 6.

If a system receives widespread use, the numbers of users, items and ratings are likely to be continually growing and to become extremely large. Hence, we are particularly interested in the scalability of Collaborative Recommenders. It may be that improvements in, for example, accuracy are not justified if they result in substantially longer response times.

In this vein, we will also report the worst-case time complexity and space complexity (which will include the size of the model, if there is one) for the on-line components of the algorithms. And, we will also discuss how easy it is to introduce new users, new items and new ratings. Some model-based approaches have particular problems with new data. There may be no efficient, incremental way of revising the model to take account of the new data. During the period between the arrival of the new data and the periodic re-generation of the model, the value of the system to its users may be diminished.

3 Exhaustive Recommender

The Exhaustive Recommender is a memory-based system whose performance is used as a baseline in comparisons with the other algorithms. We described it briefly in the previous section. Here we present some of the details.

In [5], the Exhaustive algorithm was subjected to an extensive empirical investigation using the MovieLens dataset. The decisions that we make below about the details of the algorithm (the choice of Pearson correlation as the similarity formula, the weighting factor of 50, the choice of 20 as the size of the neighbourhood and the choice of prediction formula) are based on the best results reported in [5]. We use the exact same settings in all our other experiments using different algorithms. Because of the time it would take, we have not verified that these are the best settings for these other algorithms. These other algorithms generally out-perform the Exhaustive algorithm. So the effect of not finding the best settings is to understate the extent to which they out-perform the Exhaustive algorithm. More questionably, we use the same settings in our experiments on the PTV dataset.

The Exhaustive algorithm works as follows:

- The similarity $w_{u_a,u}$ between the active user u_a and each other user $u \neq u_a$ *who has rated* i is computed using Pearson Correlation [5].

$$w_{u_a,u} \hat{=} \frac{\sum_{i=1}^{m}(r_{u_a,i} - \bar{r}_{u_a})(r_{u,i} - \bar{r}_u)}{\sigma_{u_a}\sigma_u} \times \frac{s}{50} \qquad (1)$$

  $\bar{r}$ denotes a mean value and σ denotes a standard deviation, and these are computed on co-rated items only. If a user has given the same rating to all the co-rated items, his/her standard deviation is zero and $w_{u_a,u}$ is then defined to be zero (J.Herlocker, pers. comm. 2002). Following [5], the Pearson Correlation coefficient is weighted by a factor of $\frac{s}{50}$ where s is the number of co-rated items. This decreases the similarity between users who have fewer than 50 co-rated items (who, even if their ratings are very similar, are likely to be poor predictors).
- After computing the similarity between u_a and each other user u who has rated i, the 20 nearest neighbours are selected, i.e. the 20 for whom $w_{u_a,u}$ is highest.
- The predicted rating $p_{u_a,i}$ is computed from the neighbours' ratings of i as follows:

$$p_{u_a,i} \hat{=} \bar{r}_{u_a} + \frac{\sum_{u=1}^{k}(r_{u,i} - \bar{r}_u)w_{u_a,u}}{\sum_{u=1}^{k} w_{u_a,u}} \qquad (2)$$

  where k is the number of neighbours (in our case, 20). This is essentially a weighted average of the neighbours' ratings for item i (weighted by their similarities). If $p_{u_a,i}$ goes out of the range of legal ratings, it is rounded to the nearest end-point of the range (J.Herlocker, pers. comm. 2002).

In fact, we improve the prediction efficiency of this algorithm by doing some off-line computation. We pre-compute a matrix of user-user similarities (Equation 1). This means that, on-line, we need only find the 20 nearest neighbours who have rated i and compute their prediction for i using Equation 2.

The accuracy, coverage and efficiency results for running this algorithm (and our other two algorithms) on the two datasets are shown in Figures 1– 6. Figures 1 and 4 show that accuracy is good when the dataset becomes less sparse. But Figures 2 and 5 show that coverage is not maximal. The real problem with this algorithm, however, shown by Figures 3 and 6, is that it does not scale well. In these two plots, we are counting the following operations: we count as one operation the check to see whether the user has rated i; then we count all the comparisons needed to find the 20 nearest neighbours; and then we count each rating that we aggregate when making the final prediction.

The algorithm must compare the active user with all other users, of which there are n, checking each to see whether they have rated i. Finding the k nearest neighbours will require nk comparisons in the worst case. The prediction formula then requires k steps, to compute the average rating for i over the k neighbours. We are assuming here that Equations 1 and 2 are reformulated to allow incremental computation, so means and standard deviations can be computed without multiple passes through the co-rated items. The total cost is therefore $n(1 + k) + k$. Taking k to be constant, the time complexity is $O(n)$. (Without the user-user matrix, it is $O(nm)$.)

The space complexity is the cost of storing matrix r, which is $O(nm)$, plus the cost of storing the user-user matrix, which is $O(n^2)$.

Irrespective of whether a user-user matrix is pre-computed or not, this algorithm easily incorporates new users, new items and new ratings. We simply update r and, if applicable, we make incremental updates to the similarities in the user-user matrix.

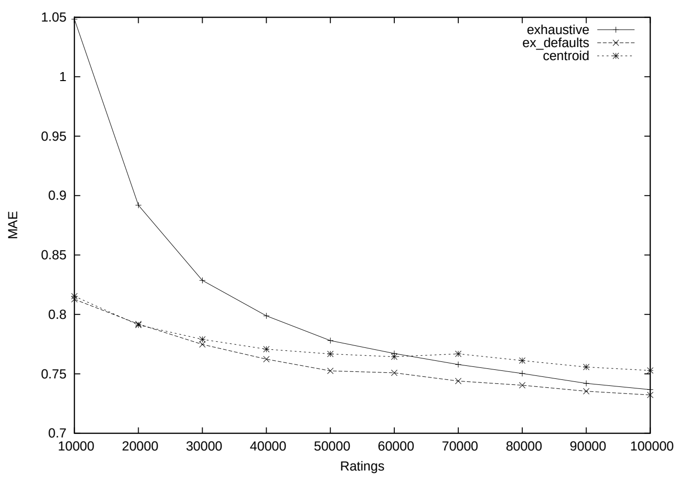

Figure 1. MovieLens Error Results

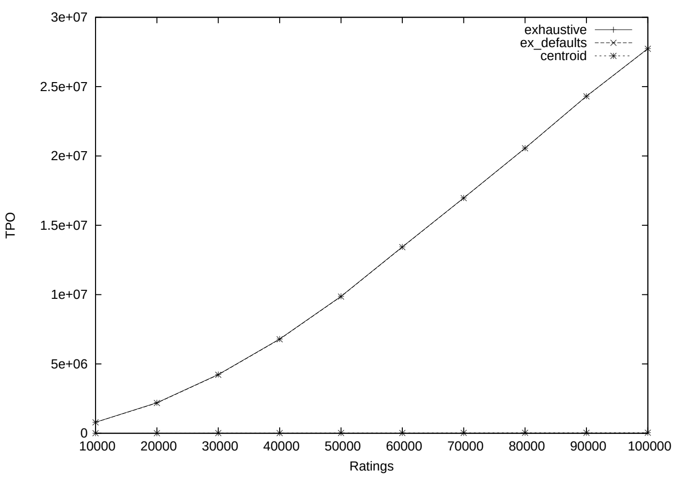

Figure 3. MovieLens Efficiency Results

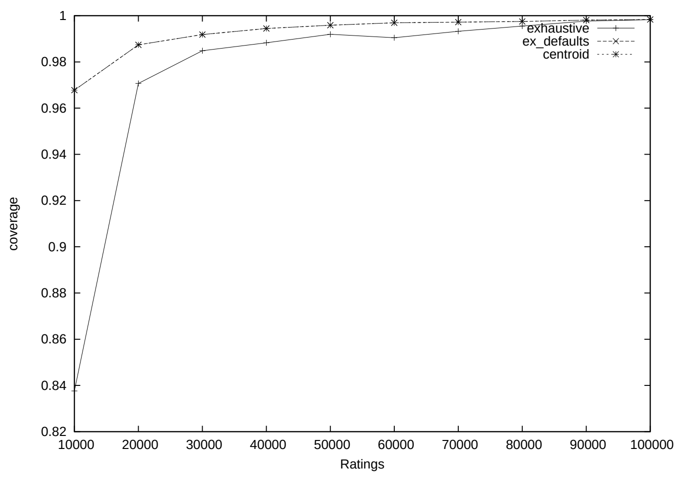

Figure 2. MovieLens Coverage Results

4 Exhaustive with Defaults Recommender

There are occasions when the Exhaustive Recommender cannot make a prediction. We can increase coverage by providing a strategy for making default predictions in such cases. A simple strategy is, when no prediction can be made, predict i's average rating in the dataset.

But, in fact, we have found that both coverage *and* accuracy can be increased if the default strategy is invoked more often. It is well-known that non-personalised predictions can be more accurate in cases where the recommender does not have enough ratings to provide an accurate personalised recommendation. Our Exhaustive with Defaults Recommender makes default predictions whenever the number of neighbours who have rated i falls below a minimum threshold. This, of course, includes the case where no neighbour has rated the item. In our experiments, the threshold we have found to be best is actually 20. In other words, if the algorithm fails to find a full set of 20 neighbours who have rated i, it uses a default prediction.

The default prediction for i, def_i, is defined as follows:

$$def_i \hat{=} avg_i(U) \tag{3}$$

where for set of users S,

$$avg_i(S) \hat{=} \begin{cases} \frac{\sum_{\{u \in S : r_{u,i} \neq \bot\}} r_{u,i}}{|\{u \in S : r_{u,i} \neq \bot\}|} & \text{if } \exists u \in S : r_{u,i} \neq \bot \\ \bot & \text{otherwise} \end{cases} \tag{4}$$

This is simply the average rating for i over all users who have rated i.

Exhaustive with Defaults is an algorithm with *maximal* coverage. The only cases in which a prediction will still fail to be made are those where *no one* in the whole dataset has rated i.

The default predictions (averages) can be pre-computed, and are easily updated incrementally when new ratings arrive. The on-line cost of providing any default prediction is, therefore, $O(1)$. So, the overall time and space worst case complexities of Exhaustive with Defaults are no different from those of Exhaustive.

The advantage of including results for this algorithm in this paper is that this algorithm provides fairer accuracy and efficiency comparisons with our RecTree Centroid Recommender (Section 6). Both Exhaustive with Defaults and RecTree Centroid have maximal coverage. This means that accuracy and efficiency results can be compared for exactly the same sets of predictions.

We turn now to the RecTree Centroid algorithm. We first explain the way we cluster the data, and then we explain the RecTree Centroid Recommender itself.

5 The Clustering Algorithm

The clustering algorithm which we have chosen is called RecTree. It builds a binary tree of clusters; it was previously used to cluster users in [3]. RecTree recursively invokes the k-means clustering algorithm. In k-means, elements are repeatedly assigned to clusters on the basis of their similarity to the cluster centre. We use Pearson Correlation, Equation 1, to compute similarities. The centres are initially

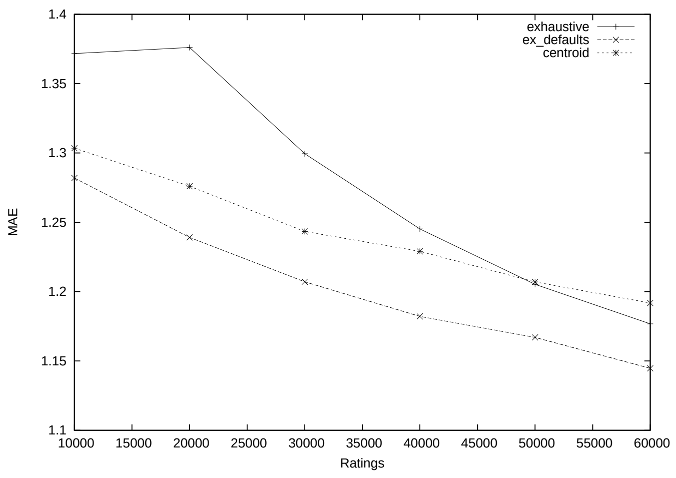

Figure 4. PTV Error Results

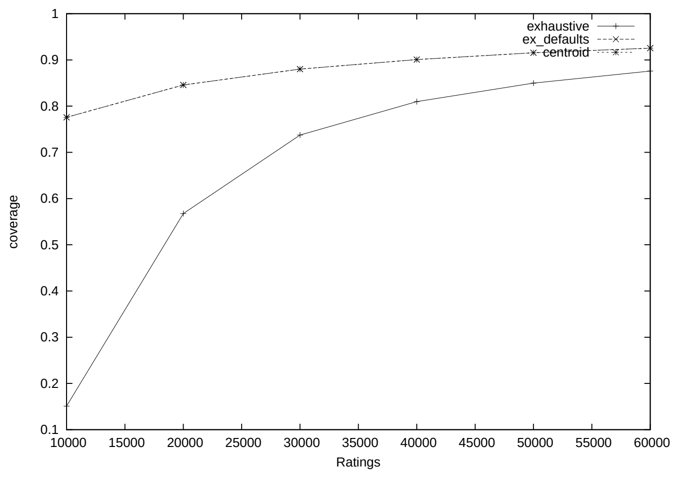

Figure 5. PTV Coverage Results

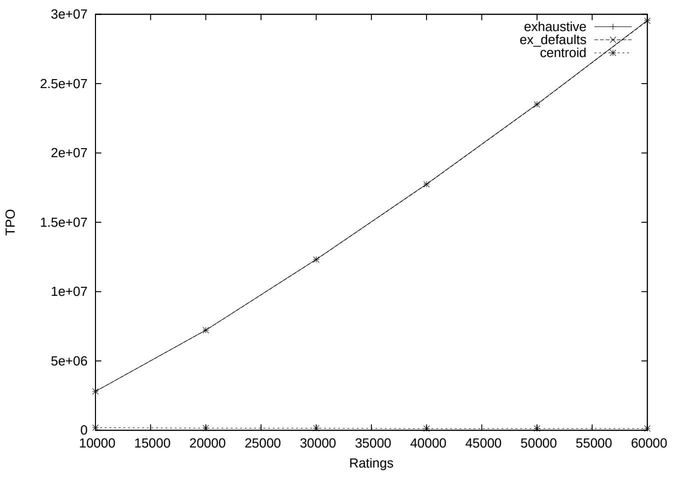

Figure 6. PTV Efficiency Results

seeds, provided as parameters. As the algorithm iterates, the centres become the centroids of the existing clusters, where a centroid is a new 'dummy' element formed by averaging ratings within the cluster. Specifically, the centroid of cluster c that contains users U_c gives rating $cent_{c,i}$ to item i:

$$cent_{c,i} \hat{=} avg_i(U_c) \quad (5)$$

which uses Equation 4 to compute the average rating, but we compute this only for the users in cluster c.

We consider k-means to have converged when the size of all clusters is the same on two successive iterations. We found empirically that convergence usually takes about 15 iterations. To ensure that the algorithm always terminates, even on pathological data, we imposed a complementary stopping criterion in the form of an iteration limit of 20.

The RecTree algorithm calls k-means to split successive datasets into child clusters. It returns a binary tree of clusters, where the root represents the whole dataset. The algorithm splits the dataset if its

Algorithm 1 RecTree(*parent, elements*) [3]

```
$v \leftarrow$ new node having *elements* as its contents
make $v$ a child of *parent*
**if** *elements* size $>$ max. leaf cluster size and no. of internals $<$ max. no. of internals **then**
  *centroid* $\leftarrow$ centroid of *elements*
  *seed1* $\leftarrow$ the member of *elements* with min. correlation with *centroid*
  *seed2* $\leftarrow$ the member of *elements*$\setminus\{$*seed1*$\}$ with min. correlation with *seed1*
  *clusters* $\leftarrow$ $k$-means(*elements*, $\{$*seed1*, *seed2*$\}$)
  **for all** $c$ in *clusters* **do**
    *c-elements* $\leftarrow$ elements in cluster $c$
    RecTree($v$, *c-elements*)
  **end for**
**end if**
**return** $v$
```

size is greater than a pre-specified maximum leaf cluster size. It selects seeds using the 'two mutually well separated centres policy' [3]. The first seed is the element within the dataset whose distance is greatest from the dataset centroid. The second seed is the element which is most distant from the first seed.

RecTree terminates when the sizes of all leaf nodes are less than or equal to the maximum leaf cluster size and so no more subdivision is required. In some cases, however, when data is not suitably distributed, growth in the tree can be 'lopsided'. This is where a very small and a very large cluster are created at each invocation. In this case we prevent the construction algorithm from over-partitioning the data by limiting the number of internal nodes (in our case to 2000). Because of this, the algorithm makes no guarantee about the size of any leaf cluster; but if the algorithm terminates normally, leaf clusters will have at most the maximum leaf cluster size number of elements.

For each user, we store a reference to the leaf cluster to which s/he is ultimately assigned. This allows $O(1)$ access to that cluster. For each cluster (including interior clusters), we store the cluster's

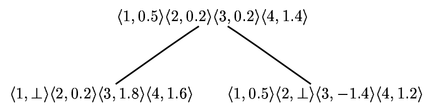
$\langle 1, 0.5\rangle\langle 2, 0.2\rangle\langle 3, 0.2\rangle\langle 4, 1.4\rangle$
$\langle 1, \perp\rangle\langle 2, 0.2\rangle\langle 3, 1.8\rangle\langle 4, 1.6\rangle$
$\langle 1, 0.5\rangle\langle 2, \perp\rangle\langle 3, -1.4\rangle\langle 4, 1.2\rangle$

Figure 7. Extract from a RecTree

centroid (Equation 5) to give $O(1)$ access time to this information.

We can obtain some idea of the cost of walking and storing the RecTree if we make some assumptions about the shape of the tree that gets built. Let us idealise and assume that the two children of a node share equally the users of their parent, i.e. the two children are the same size, which is half their parent size. At any level of the tree, this will give clusters whose sizes differ by at most one. For n users and a maximum leaf cluster size of max, the height of the tree h is $\lceil \log_2 n - \log_2 max \rceil$, which is $O(log_2 n)$. There are $2^h + 2^{h-1} + \ldots + 1$ nodes in the tree, which, given that h is $O(\log_2 n)$, is $O(n)$. There are $\lceil n/max \rceil$ (or, equivalently, 2^h) leaf clusters.

6 RecTree Centroid Recommender

The RecTree Centroid Recommender is a model-based recommender. Predictions are obtained from the RecTree, which is used as a user-based model.

Suppose the active user u_a is a member of leaf cluster c. An initial prediction of u_a's rating for item i is the average rating of i for the users in cluster c, as given by the centroid. However, from Equations 5 and 4, we know that if no one in the cluster has rated i, then $cent_{c,i}$ will be $\perp$. In this event, the RecTree Centroid Recommender visits the parent of cluster c and returns the rating for i given by the parent's centroid (if this is not $\perp$). The algorithm climbs the RecTree until it finds a centroid whose rating for i is not $\perp$ or until it reaches the root, in which case it returns the rating contained in the centroid for the root (whether this be $\perp$ or not).

For example, suppose that the active user belongs to the right-hand leaf cluster in Figure 7. The ratings stored at this node are the average ratings from the users that belong to this cluster. In particular, for items 1, 3 and 4, the average ratings are 0.5, -1.4 and 1.2 respectively; no-one in this cluster has rated item 2. If we want to predict the active user's rating for item 4, for example, we simply return the rating stored at this node, i.e. 1.2. If we want to predict a rating for item 2, we ascend the tree until we find a node at which there is a rating for item 2 (or until we reach the root). Recall that a parent node in the RecTree represents the union of the users associated with its two child nodes. There is thus a greater likelihood that the parent's centroid will have a rating for i that is not $\perp$. In Figure 7, for example, the parent has a rating for item 2 because some of the users associated with the left-hand child have rated item 2; so the prediction is 0.2. The root node represents all users and so the root node's centroid will only have no rating for i if no one in the entire system has rated item i.

It might be argued that, while this algorithm is certainly a collaborative one (since the preferences of other users are used to construct the RecTree), it is a less *personalised* recommender: all users in the same cluster receive the same predictions for items they have not rated.

For the MovieLens dataset, we found experimentally that a maximum leaf cluster size of 300 gave the best results, and so (although it may not be best for PTV) this is what we used for both MovieLens and PTV in Figures 1– 6.

Figures 1 and 4 show that RecTree Centroid actually has lower error than the Exhaustive Recommender for the sparsest datasets and only has higher error as ratings grow beyond a certain point; it is never more accurate than Exhaustive with Defaults but it is still highly competitive. The use of z-scores makes a big difference: when we used raw ratings (not plotted), error was much higher.

Coverage (Figures 2 and 5) is maximal, the same as Exhaustive with Defaults. In both algorithms, the only circumstance in which a prediction will not be made is if no user has rated item i.

But RecTree Centroid is very efficient. In Figures 3 and 6, the plot (which counts tree node accesses) lies slightly above the x-axis. It takes constant time to access a user's leaf cluster. Then, in the worst case, the algorithm climbs the whole tree, from the leaf cluster to the root. So the worst case time is governed by the height of the tree, $\lceil \log_2 n - \log_2 max \rceil$. Importantly, then, prediction time is *independent* of the number of ratings (and items), and it is less than linear in the number of users ($O(log_2 n)$).

Space requirements are higher than for the Exhaustive algorithms, but still acceptable. We must store the RecTree ($2^h + 2^{h-1} + \ldots + 1$ nodes, where h is the height of the tree). Each node stores just its centroid, containing one rating per item, and so is of size m. We must also store each user with a reference to his/her leaf cluster.

New users, new items and new ratings can all be accommodated efficiently on-line. A new user must be inserted into the most appropriate leaf cluster. We would do this by walking down the tree from the root to a leaf, at each step selecting the child for which the correlation between the new user's ratings and the child's centroid is the greater. This will take time bounded by hm.

A new item simply requires that each centroid's vector of ratings be extended in length. A new rating (for an existing user and item) requires updates to the centroid rating for that item in the user's leaf cluster and its ancestors. These updates can be done incrementally. However, this will not have any affect on the predictions we would make for the user who supplied the new rating: s/he is still in the self-same cluster. To counter this, if a user supplies a large number of new ratings, then it may be better to delete the user from his/her leaf cluster (incrementally updating centroids in the tree appropriately), and then treat this user as a new user. This gives the possibility, if this user's ratings are now much changed, that s/he will be placed into one of the other clusters. Of course, this is still not as good as re-generation of the tree, since it assumes that the existing clusters are basically correct. Periodic re-generation will still be necessary.

7 Comparisons with Related Work

We look in particular here at work that uses clustering for Collaborative Recommenders. In [2], we claimed that clustering can be used for two different purposes: *partitioning* and *grouping*. In partitioning, a dataset is divided so that search can be confined to one of the partitions; in grouping, a dataset is divided and then a composite object (a 'super-user' or 'super-item') acts as proxy for the members of the group.

In [2], we described the Clustered Users algorithm, which is a neighbourhood-based algorithm that uses the RecTree to *partition* users. Its search for neighbours is confined to users who are in the same leaf cluster as the active user. Hence, all equations in Section 3 are used, but with u ranging only over members of c, the active user's leaf cluster.

In [2], we describe how Clustered Users makes non-personalised predictions in certain cases. When we construct the RecTree, it is possible that some leaf clusters will have very few members; if they have fewer than a certain threshold (in our case 10), we call that node of the tree an *outlier node*. During prediction, if the user's leaf cluster is an outlier node, then we ascend the RecTree *one* level and use the centroid rating from the parent. This is similar to the recommender algorithm described by Chee [3]. Our results can be seen in [2]. Clustered Users is much less accurate, has much lower coverage but is more efficient than Exhaustive. However, Clustered Users is not competitive with RecTree Centroid: its accuracy, coverage and efficiency are worse across all dataset sizes.

We have recently investigated a variant of Clustered Users. We abandon the idea of outlier nodes, and instead we incorporate ideas from Exhaustive with Defaults and RecTree Centroid. Specifically, if the number of neighbours (still drawn from the active user's leaf cluster) who have rated i falls below a minimum threshold, we invoke the RecTree Centroid method of finding a rating, i.e. we climb the RecTree until we reach either the root or a node whose centroid includes a rating for i other than $\perp$. We have not published the results for this algorithm because, unfortunately, it is not competitive. On the positive side, its coverage is maximal. But, its accuracy is worse than Exhaustive with Defaults and RecTree Centroid for all dataset sizes. And, while it remains more efficient than Exhaustive algorithms, the fact that it is still memory-based means that it is much slower than RecTree Centroid.

Breese et al. describe a model-based approach in which users are clustered probabilistically [1]. A prediction takes the form of a probability for a rating. It is not possible to compare their results directly with ours because they use different datasets and different evaluation measures. One observation, however, is that their clustering algorithm performed relatively poorly compared with their Bayesian network and memory-based approaches. Since our clustering method does well against our memory-based method, this gives us reason to hope that we have found a better way of using clustered data.

Fisher et al. use just the k-means algorithm to *group* users [4]. They store the centroids of the clusters, but they have no tree as we do. To make predictions, their Clustered Pearson Predictor treats the centroids as super-users and seeks neighbours only among the super-users. This was the most accurate and scalable of the algorithms they used. But, their results are given for a different dataset from ours and are not plotted for different total numbers of ratings. So, again, no direct comparison can be given. One observation is that their dataset contained 60000 users for which they created 5000 clusters; to find neighbours therefore requires at least 5000 operations. If we had built a RecTree for this dataset (assuming a maximum leaf cluster size of 300 again), the height of the tree, which determines prediction effort, would have been $\lceil \log_2 60000 - \log_2 300 \rceil \approx 7$.

It is possible, of course, to cluster *items* instead of, or as well as, users. O'Connor & Herlocker *partition* items using Pearson correlation [7]. To make a prediction for item i, the Exhaustive algorithm is applied only to co-rated items from i's partition. O'Connor & Herlocker expected accuracy and coverage to be higher, but found experimentally that this was not so, irrespective of which of several clustering algorithms they used. Efficiency, of course, improves but continues to take time proportional to the number of users.

In [2], we describe an algorithm we call Clustered Items in which we *group* items, to produce super-items. This gives a denser dataset. Applying the Exhaustive algorithm to this denser dataset gives accuracy that is only a little worse than Exhaustive and coverage that is slightly higher than Exhaustive. Coverage is still not maximal although one could introduce default predictions to obtain maximal coverage. As one would expect with a denser dataset, the algorithm is far less efficient than Exhaustive.

It is possible to cluster *both* users and items. In [2], we plotted results for a Dual Clustered algorithm. We *partitioned* users first and then *grouped* items into super-items. We chose this ordering to avoid the reduction in accuracy that would occur if we were to cluster users after their ratings for items had been 'collapsed' into ratings for super-items. To make a prediction, the algorithm first finds the item's super-item and then applies neighbourhood-based methods to the user's cluster to make a prediction for the super-item. Unfortunately, this was our least efficient and least accurate algorithm.

In [8], Ungar & Foster cluster users based on items, and then items based on users; then users are clustered based on item clusters, and items based on user clusters; and then this is repeated three times. The item clustering can be done from data in the ratings matrix r but, in their experiments on real CD purchase data, item clustering was done in a content-based way, by CD artiste. Experimental prediction results are not given.

In contrast, Kohrs & Merialdo cluster users and items independently into two cluster hierarchies [6]. The hierarchies are produced in a way that is highly similar to the way we build our RecTrees. Their prediction formula is very different: they use a weighted sum of the centroids of all nodes on the path in the user's hierarchy from the user's leaf cluster to the root and all nodes on the path in the item hierarchy from the item's leaf cluster to the root. The weights are based on cluster distortion, this being the sum of the distances between ratings and the centroid rating. Results are presented for a different dataset from ours. We regard a proper comparison between their algorithm and ours to be an objective of our future work.

In conclusion, we have proposed two new algorithms, Exhaustive with Defaults and RecTree Centroid. Both have maximal coverage. Their relative usefulness, therefore, depends on the trade-off between accuracy and efficiency: Exhaustive with Defaults is marginally more accurate, while RecTree Centroid is hugely more efficient. In the domains in which Collaborative Recommenders are used, there may be tens of thousands, if not millions, of items and users. In domains with these characteristics, RecTree Centroid has the greater promise: it is likely to meet response time requirements, while still making reasonably accurate predictions.

REFERENCES

[1] J. S. Breese, D. Heckerman, and C. Kadie, 'Empirical analysis of predictive algorithms for collaborative fi ltering', in *Procs. of the 14th Annual Conference on Uncertainty in Artificial Intelligence*, pp. 43–52, (1998).

[2] D. Bridge and J. Kelleher, 'Experiments in sparsity reduction: Using clustering in collaborative recommenders', in *Procs. of the Thirteenth Irish Conference on Artificial Intelligence and Cognitive Science*, eds., M.O'Neill, R.F.E.Sutcliffe, C.Ryan, M.Eaton, and N.J.L.Griffi th, pp. 144–149. Springer, (2002).

[3] S. H. S. Chee, *RecTree: A Linear Collaborative Filtering Algorithm*, Master's thesis, Simon Fraser University, 2000.

[4] D. Fisher, K. Hildrum, J. Hong, M. Newman, M. Thomas, and R. Vuduc, 'Swami: A framework for collaborative fi ltering algorithm development and evaluation', in *Procs. of SIGIR*, pp. 366–368, (2000).

[5] J. L. Herlocker, *Understanding and Improving Automated Collaborative Filtering Systems*, Ph.D. dissertation, University of Minnesota, 2000.

[6] A. Kohrs and B. Merialdo, 'Clustering for collaborative fi ltering applications', in *Procs. of CIMCA'99*. IOS Press, (1999).

[7] M. O'Connor and J. Herlocker, 'Clustering items for collaborative fi ltering', in *Procs. of the ACM SIGIR Workshop on Recommender Systems*, (1999).

[8] L.H. Ungar and D.P. Foster, 'Clustering methods for collaborative fi ltering', in *AAAI Workshop on Recommendation Systems*, (1998).