



Inverted File Search Algorithms for Collaborative Filtering

Rickard Cöster
Stockholm University, DSV
Forum 100, SE-164 40 Kista, SWEDEN
rick@dsv.su.se

Martin Svensson
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, SWEDEN
martins@sics.se

ABSTRACT

This paper explores the possibility of using a disk based inverted file structure for collaborative filtering. Our hypothesis is that this allows for faster calculation of predictions and also that early termination heuristics may be used to further speed up the filtering process and perhaps even improve the quality of the predictions. In an experiment on the EachMovie dataset this was tested. Our results indicate that searching the inverted file structure is many times faster than general in-memory vector search, even for very large profiles. The Continue termination heuristics produces the best ranked predictions in our experiments, and Quit is the top performer in terms of speed.

Categories and Subject Descriptors

H.3.3 [Information Search and Retrieval]: Information filtering; H.3.4 [Systems and Software]: Performance evaluation (efficiency and effectiveness)

General Terms

Algorithms, Performance

Keywords

Collaborative Filtering, Inverted Files, Performance

1. INTRODUCTION

The Internet is an ever-expanding repository of information that needs to be filtered and navigated. To meet this need collaborative filtering has emerged as a complementary method to traditional filtering methods such as content based filtering.

This paper explores the possibility to use an inverted file structure for collaborative filtering. Our hypothesis is that this structure allows for faster calculation of predictions and also that pruning strategies (Quit and Continue) from information retrieval can be used to further speed up the filtering process and perhaps even improve their quality.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGIR'02, August 11-15, 2002, Tampere, Finland.
Copyright 2002 ACM 1-58113-561-0/02/0008 ...\$5.00.

We have two reasons for this. Firstly, matching user profiles in a collaborative filtering system can be very expensive. Secondly, if we are able to use a structure that allows us to access all user preferences stored directly from disk it is possible to maintain a much larger set of users and titles.

In order to evaluate this, we conducted a series of experiments on a large dataset. In these experiments, we compare inverted file search with in-memory vector search both in terms of prediction accuracy and neighbourhood formation time.

The rest of this paper is organised as follows. In section 2 we review collaborative filtering. In section 3 we express four common correlation algorithms in inverted form. The file organization we selected for implementing the inverted files is briefly described in section 4. In sections 5 and 6 we describe the experimental setup and our main results.

2. COLLABORATIVE FILTERING

The goal for a collaborative filtering system is to predict a user's preference regarding some set of titles: books, news articles, or perhaps radio channels. To do this, a collaborative filter use ratings from a database of users (either from all users or a subset of the users in the database). The filter can then be used in either of two ways: (1) users can explicitly ask what the system believes they would think about a certain title, or (2) users can ask the system to return a ranked list of titles the system thinks they like.

Collaborative filtering allows us to tackle three drawbacks found in traditional filtering methods:

- titles must be parsable, e.g. music and movies can not be searched for [13]
- it is difficult to achieve serendipity: content-based systems do not typically allow for discovery of information not sought for
- content based filtering disregards qualitative aspects such as genre [7].

Combined with regular content-based techniques collaborative filtering offers a powerful way of finding and filtering information.

As with content based filtering there are a number of problems that need to be resolved: **Bootstrapping**: Since collaborative filtering relies on the ability of matching user profiles, they will work poorly when little information has been collected [12]. **Concept drift**: Over time people and information change. What was interesting yesterday may be totally irrelevant today. Collaborative filtering systems tend

to get “conservative” in the sense that once a user profile is built, it is difficult to change. **Scalability:** In systems with large number of users and titles, algorithms that scale up to that dimension are needed.

The heart of a collaborative filter, as indicated, lies in the ability to make fast and accurate predictions. Before we present how inverted files can be used as the basic structure for the collaborative filter algorithm, let us review the two dominant models for performing predictions.

2.1 Memory-based Prediction

Memory-based prediction uses a nearest-neighbour approach to find a subset of all users that have the most similar preference history as the active user [6]. When this neighbourhood is found, the similarity between the active user and a neighbour determines how much the neighbour influences the prediction for the active user. If the similarity is high, the neighbour has high influence on the final prediction. Thus, the final prediction is a weighted combination of the neighbours’ preferences. Memory-based predictions have been used in systems such as GroupLens [8] and Ringo [13].

The advantage of the memory-based scheme over other methods is that its structure is dynamic and immediately reacts to changes in the user database. Every new rating added to the user database will be included in the neighbourhood search. The reason for this is of course that similarities between users are calculated “in memory” when needed. This property is also the potential drawback of the method. When user profiles are matched against each other every time a prediction is needed, the process can be extremely slow. This would not only take time, but also require a large amount of memory. In the general case this problem is solved by only keeping parts of the user database in memory and treat that as the set of users to draw the neighbourhood from (sampling), or to precompute the neighbourhoods.

2.2 Model-based Prediction

A common technique in machine learning to predict a value is to build a model of some input data and then run new data through that model to get a predicted output. This can also be applied to collaborative filtering. The user database is run through an algorithm that builds a model of that data in terms of the probability that a user would like a given title depending on what she has done in the past. For example, in a Bayesian network the titles would represent nodes where the state of each node would represent possible vote values [3]. Various other model-based techniques have been applied in collaborative filtering, e.g. Latent Semantic Indexing [11], Singular Value Decomposition [2] and Mixture Models [9].

In terms of predictive accuracy the model-based schemas perform as well as memory-based prediction [3]. On the other hand, the model-based prediction does not suffer from the performance and memory bottlenecks found in memory-based prediction. Once a good model has been found a predication takes a small amount of time and resources to calculate. The downside of the scheme is of course the inherent static structure of it. Once the model has been built it is difficult to update it without rebuilding it. In dynamic domains where users and votes are constantly changing the model could soon become inaccurate.

3. INVERTED SEARCH

The reason for using inverted files in information retrieval is the observation that a user query usually contains a small percentage of the total number of terms in the document collection. In collaborative filtering a user’s votes (or profile) can be seen as a document, which in itself can be used as a query. However, there are some differences between a document and a profile. If the user, as in a movie recommender system, explicitly expresses the votes, the user’s profile will certainly grow over time and perhaps contain several hundreds or even thousands of votes. In the case of implicit voting, the profile may grow even faster when, for example, each web page visit or news article read render a vote that is stored in the profile.

Since the user profiles are expected to grow over time, it may seem unlikely that inverted files are suited for collaborative filtering. We argue quite the opposite, for two main reasons: the first being that inverted file search is well researched and handles scalability very well. The second is that the inverted file structure makes it easy to employ heuristic stop conditions, so that it is not necessary to investigate all inverted lists in a neighbourhood search and thus further speed up execution.

The basic algorithm for inverted file search scans one inverted list at a time [15]. During this scan, accumulators are stored in main memory for holding partial sums of scores or weights. In our case, the partial sums will be the partial correlations or similarities between the active user and all other users. When all inverted lists are processed, the accumulators contain the complete sums. Usually there are one or more arrays of static weights, such as normalizing factors, which are combined with the accumulators to produce the final score for a user. The final score is calculated for all non-zero accumulators, and users with top scores are returned.

One of the most commonly used algorithms for calculating neighbours in memory-based predictions is Pearson correlation. Various extensions have been made to the basic correlation [3]. We will demonstrate, in sections 3.1 through 3.4, how these algorithms may be expressed in a form suitable for inverted retrieval.

3.1 Pearson Correlation

The Pearson correlation between users a and i is defined as

$$w(a, i) = \frac{\sum_j (v_{a,j} - \bar{v}_a)(v_{i,j} - \bar{v}_i)}{\sqrt{\sum_j (v_{a,j} - \bar{v}_a)^2 \sum_j (v_{i,j} - \bar{v}_i)^2}} \quad (1)$$

where v_a is the profile for user a , $v_{a,j}$ is that user’s vote for title j and $\bar{v}_a$ is the mean value of the votes for user a . The index j runs over the intersecting titles in the two profiles, i.e. $j \in \{v_a \cap v_i\}$.

To apply inverted search to correlation, it is necessary to keep three different accumulator structures in memory: one for the sum in the nominator, and two for the sums in the denominator. Call these variables SAI , SAA and SII respectively. In addition, the mean values for each user must be easily accessible, and these are stored in the static array *MEANS*. Let N denote the number of users in the database. Algorithm 1 will then compute the top K neighbours to user a . The sorting at step 15 can e.g. be performed by min-heaps [15].

Algorithm 1 Inverted correlation neighbourhood search

Input is the user profile V_a , max number of neighbours K and array of means $MEANS$. Output is array of nearest neighbours.

```

1: Allocate accumulators  $SAI$ ,  $SAA$  and  $SII$  of size  $N$ 
2: Allocate array  $TOP$  of size  $K$  for nearest neighbours
3: for all  $j \in v_a$  do
4:   locate inverted list  $L$  for title  $j$ 
5:   for each user  $u$  and vote  $v$  in  $L$  do
6:      $SAI[u] = SAI[u] + (v_{a,j} - \bar{v}_a) * (v - MEANS[u])$ 
7:      $SAA[u] = SAA[u] + (v_{a,j} - \bar{v}_a)^2$ 
8:      $SII[u] = SII[u] + (v - MEANS[u])^2$ 
9:   end for
10: end for
11: for all  $u \in SAI$ , such that  $SAI[u] \neq 0$  do
12:    $corr = SAI[u] / \sqrt{(SAA[u] * SII[u])}$ 
13:   if  $corr > TOP[K - 1]$  then
14:     add  $(u, corr)$  to  $TOP$ 
15:   restore  $TOP$  to sorted order
16: end if
17: end for
18: return  $TOP$ 

```

3.2 Inverse User Frequency

An extension to the basic Pearson correlation algorithm is inverse user frequency [3], which assigns lower weights to titles that have a larger number of votes. The assumption is that titles that are viewed by a large number of users are not as useful in capturing the similarity between two users as titles that have been viewed by fewer users. Inverse user frequency is similar to the inverse document frequency weighting [10] used in text retrieval.

The inverse user frequency for title j can be defined as $f_j = \log(N/n_j)$ where n_j is the number of users that have voted for title j . This weight is calculated for each title in the data set.

For the algorithm, we use the same notation as in [3]. The inverted search algorithm requires six accumulators for the different sums $\sum_j f_j$, $\sum_j f_j v_{a,j} v_{i,j}$, $\sum_j f_j v_{a,j}^2$, $\sum_j f_j v_{i,j}^2$, $\sum_j f_j v_{a,j}^2$ and $\sum_j f_j v_{i,j}^2$. Furthermore, the inverse frequencies should be kept in a static array IUF , which at index j holds the value $\log(N/n_j)$. The inner loop of the search algorithm is then similar to that of Algorithm 1.

3.3 Default Voting

Default voting is computed between two users a and i in the union of their votes. If both users have not voted for title j , a default vote d is used for the missing vote. The default vote is selected globally for the whole data set, as a neutral or slightly negative vote. Another parameter k estimates the number of titles two users have not seen or voted on but would agree on.

Let n be the number of titles that both user a and i have voted for. Default voting is then

$$w(a, i) = \frac{A_1 - B_1}{\sqrt{U_1 V_1}} \quad (2)$$

where

$$\begin{aligned}
A_1 &= (n+k) \left(\sum_j v_{a,j} v_{i,j} + k d^2 \right) \\
B_1 &= \left(\sum_j v_{a,j} + k d \right) \left(\sum_j v_{i,j} + k d \right) \\
U_1 &= (n+k) \left(\sum_j v_{a,j}^2 + k d^2 \right) - \left(\sum_j v_{a,j} + k d \right)^2 \\
V_1 &= (n+k) \left(\sum_j v_{i,j}^2 + k d^2 \right) - \left(\sum_j v_{i,j} + k d \right)^2
\end{aligned}$$

Recall that the inverted search finds the intersection of the votes of the active user and all other users. This leads to a slight complication when inverting default voting, since it is calculated in the union of the user's votes, i.e. now $j \in \{v_a \cup v_i\}$. To remedy this, we need to express the sum over union of titles in terms of the sum of the intersecting titles. The sum in A_1 can be written as

$$\begin{aligned}
\sum_j v_{a,j} v_{i,j} &= \sum_m v_{a,m} v_{i,m} + \\
&+ d \left(\sum_s v_{a,s} - \sum_m v_{a,m} \right) + \\
&+ d \left(\sum_t v_{i,t} - \sum_m v_{i,m} \right)
\end{aligned} \quad (3)$$

where $j \in \{v_a \cup v_i\}$, $s \in \{v_a\}$, $t \in \{v_i\}$ and $m \in \{v_a \cap v_i\}$.

The idea is then to accumulate the sum over m during the search and have the sum over t in a static array. The sum over s (for the active user a) may be stored in the static array or calculated during the search.

The four remaining sums for equation 2 ($\sum_j v_{a,j}$, $\sum_j v_{i,j}$, $\sum_j v_{a,j}^2$ and $\sum_j v_{i,j}^2$) can be calculated using a similar argument. Consider e.g. the sum $\sum_j v_{a,j}$, which is equal to $\sum_s v_{a,s} + d(|v_i| - |v_a \cap v_i|)$. This may be calculated if we know $\sum_s v_{a,s}$ and $|v_i|$, and calculate $|v_a \cap v_i|$ during the search. This sum and the three remaining ones are symmetrical.

Altogether, three static arrays are needed to look up the values of $\sum_t v_{i,t}$, $\sum_t v_{i,t}^2$ and $|v_i|$. Four accumulators are needed, three for the partial sums over m ($\sum_m v_{a,m} v_{i,m}$, $\sum_m v_{a,m}$, $\sum_m v_{i,m}$) and one for the number of intersecting titles $|v_a \cap v_i|$. Again, the sums over s may be stored in the static arrays or calculated during the search. We chose the latter for our experiments.

3.4 Default Voting and Inverse User Frequency

The idea of weighting titles according to their inverse user frequency can also be applied to default voting. If we use the index variables j , s , t and m as in the previous section, then default voting with inverse user frequency can be expressed as

$$w(a, i) = \frac{A_2 - B_2}{\sqrt{U_2 V_2}} \quad (4)$$

where

$$\begin{aligned}
A_2 &= (n+k) \left(\sum_j f_j v_{a,j} v_{i,j} + kd^2 \right) \\
B_2 &= \left(\sum_j f_j v_{a,j} + kd \right) \left(\sum_j f_j v_{i,j} + kd \right) \\
U_2 &= \sum_j f_j \left((n+k) \left(\sum_j f_j v_{a,j}^2 + kd^2 \right) \right. \\
&\quad \left. - \left(\sum_j f_j v_{a,j} + kd \right)^2 \right) \\
V_2 &= \sum_j f_j \left((n+k) \left(\sum_j f_j v_{i,j}^2 + kd^2 \right) \right. \\
&\quad \left. - \left(\sum_j f_j v_{i,j} + kd \right)^2 \right)
\end{aligned}$$

The sum in A_2 can be rewritten as

$$\begin{aligned}
\sum_j f_j v_{a,j} v_{i,j} &= \sum_m f_m v_{a,m} v_{i,m} + \\
&\quad + d \left(\sum_s f_s v_{a,s} - \sum_m f_m v_{a,m} \right) + \\
&\quad + d \left(\sum_t f_t v_{i,t} - \sum_m f_m v_{i,m} \right)
\end{aligned} \tag{5}$$

The remaining sums to complete equation 4 are symmetrical to $\sum_j f_j v_{a,j} = \sum_s f_s v_{a,s} + d(\sum_t f_t - \sum_m f_m)$. Again, we require the sums over t to be kept in static arrays and accumulate the sums over m . The required arrays should contain at index i the values of $\sum_t f_t$, $\sum_t f_t v_{i,t}$, $\sum_t f_t v_{i,t}^2$ and $|v_i|$. Five accumulators are needed, where four are for the sums over m ($\sum_m f_m$, $\sum_m f_m v_{a,m} v_{i,m}$, $\sum_m f_m v_{a,m}$, $\sum_m f_m v_{i,m}$). The number of intersecting titles $|v_a \cap v_i|$ is also required, to calculate $n = |v_a \cup v_i|$.

3.5 Early Termination Heuristics

As we discussed in Section 3, user profiles can grow quite large, especially in the case of implicit voting. If there are ways that would allow us to only use parts of a user profile for prediction, it would further speed up execution time.

Several heuristic stop conditions for text retrieval have been devised [15]. Some of these take advantage of the properties of the similarity function. For example, if the document weights are normalized to unit length, then it is possible to bound the accumulator weights of the cosine function [4]. Other heuristics place stop conditions on the number of traversed inverted lists or, as in our case, stop conditions on the number of users that are retrieved during the scanning of the inverted lists.

Two strategies for this are well known: Quit and Continue [15]. In both methods, the query is sorted by decreasing weight and lists are then processed in this order until the stop condition is met. When the stop condition is reached, only those accumulators that are non-zero are considered. In the Quit strategy, no more lists are processed after this point and the accumulators are used as is. The other strategy is to continue processing lists but only to update those accumulators that are already non-zero. Thus, Continue will always calculate the total weight for users that have reached into non-zero accumulators, whereas Quit may not.

We have implemented both Quit and Continue for the four algorithms described in the previous sections. The titles were weighted according to inverse user frequency.

4. FILE ORGANIZATION

We chose a Prefix B^+ -tree [5, 1] for storing our inverted index. The tree is divided into two separate files: one to store the index as minimal key separators (the index file) and one file to store the key/value pairs sorted (the value file). The key consists of a title id with the value containing either an inverted list or a pointer to a reference file that contains large inverted lists (lists larger than 256 bytes in size). The reference file was divided into 512 byte blocks and for each inverted list a pointer list was kept for tracking which blocks that list occupied.

4.1 Compression

To construct the index and the inverted lists, we used sort-based inversion [15]. The lists of run-lengths of user ids together with the vote were then compressed for each title, and stored in the reference file.

Four different compression algorithms were tried: Golomb, Elias delta, Elias gamma and variable byte coding. All these four methods are well described in the information retrieval literature [14, 15], so they will not be re-stated here. For the Golomb, Elias delta and Elias gamma codings, the vote was stored as a binary number of three bits. For variable byte coding, one byte was used.

We compressed the inverted lists for the first 60000 users that had voted for at least two titles, a total of 2805901 votes. The average number of bytes per pointer (user id, vote pair) was close to one on Golomb, Delta and Gamma coding (1.04, 1.10 and 1.11). Using variable byte coding the data was compressed to 2.05 bytes per pointer. The block file utilization was 93.7%, 93.5%, 93.2% and 96.2% respectively.

When running the experiments described in Section 5, we selected the simplest compression method, variable byte coding, since we wanted to see whether it was possible to use less complicated compression algorithms and still gain in performance by using inverted search.

4.2 Updating the Inverted Index

A collaborative filtering system will not serve its purpose if it is difficult to update it with new votes and users. In a memory-based model this is one of the greatest advantages over model-based approaches. How will the inverted file structure handle updates? The problem that needs to be solved is how to update the inverted lists.

There are two ways of handling updates, either incrementally or batch. If the index needs to be updated whenever new votes arrive, the inverted lists should be represented without the need of decompression in order to insert or delete votes. This approach has the drawback in that the list references will take up a larger amount of space. For batch updating changes would first be stored in a secondary index (i.e. in a stop-press file) and then, at some point in time, merged with the primary index.

The static arrays of weights must also be updated, and in most cases the weight depends only on local properties of the profile. It is more problematic to update the static arrays in default voting with inverse user frequency, since those values depend on global properties. If a frequency is changed for a title, all weights must be updated to reflect this change. The simplest solution is to search the inverted list for that title, and for each user in the list subtract the old value from its weight and add the new value.

5. EXPERIMENTS

To evaluate our hypothesis, we conducted a set of experiments. We measure the elapsed time to perform neighbourhood search and predictive accuracy.

We compare the inverted algorithms against in-memory vector searching. The user profiles are implemented as linked list, i.e. sparse vectors. Each object in the profile contains the title id and the vote, and each user profile is sorted on ascending title id. The different correlation algorithms were implemented as list intersection (sections 3.1 and 3.2) or list union (sections 3.3 and 3.4). The mean value was pre-computed for each profile.

5.1 Data Set and Experimental Setting

The largest available public dataset, EachMovie, was used for testing our algorithms. This set was also used in [3]. The dataset contains roughly 2.8 million explicit votes (0...5) for 61265 users on 1628 different movies. The experimental setting was then as follows:

- We cleaned the data so that if a user had duplicate votes for a title, we used the most recent.
- Users that had voted for less than 2 titles were removed, leaving a total of 60087 unique users.
- We evaluated the algorithms on the *AllBut1* protocol, meaning that for each user we held out a single vote that should be predicted on the basis of all the other votes in the profile.
- We selected the size of the training set to 55000 users and the test set to the remaining 5067 users.
- For each experiment, we ran 4 different implementations on the same data: inverted search, inverted search using Quit, inverted search using Continue and pure in-memory vector search, totalling 16 runs for an experiment. Each experiment was then performed 10 times.
- We limited the neighbourhood size to no more than 50, which has been found to be of reasonable size for movie data [6]. For Quit and Continue we set the threshold at 10000 users.

5.2 Metrics

To measure time we used two different metrics. The time for individual neighbourhood formations were plotted against the number of titles in the profile. The reason for this being that with larger profiles a neighbourhood takes longer time to calculate. In the case where there were many profiles of the same size, we used the mean time of profiles of that size. We compared the in-memory vector search with the inverse model using the basic Pearson correlation. Both these algorithms have an equal set of pre-calculated values: the inverse algorithm does not gain anything by performing pre-calculations.

We also measured mean neighbourhood formation time: how long on average the algorithm takes to form a neighbourhood.

For predictive accuracy, we use mean absolute error (MAE) for measuring single predictions. MAE calculates the absolute difference between the prediction and the actual vote, averaged over all predictions.

Table 1: Mean neighbourhood formation time

	CORR	IUF	DEF	DEFIUF
Algorithm				
VEC	664	727	1245	1420
INV	206	260	256	293
QUIT	54	52	55	63
CONT	256	308	251	271

For evaluating the ranked list we use the rank score (RS) measure developed in [3], where we used a half-life of 5. RS estimates the expected utility of a ranked list, which is taken to be the utility of a title times the probability that the user will view it. The utility of a title is in this case the difference between the vote and the default vote, and the probability than a user will view a title is an exponentially decreasing function of the position in the list.

6. RESULTS

In this section we describe the main results of applying inverted file search to collaborative filtering. We measure neighbourhood formation time in section 6.1 and predictive accuracy in section 6.2.

In Tables 1 through 3, the rows labelled INV and VEC give the results for the algorithm run on inverted search and in-memory vector search. In terms of accuracy, these two should produce the same result. The QUIT and CONT rows are the results of applying the early termination heuristics. The correlation algorithms are labelled CORR (Correlation), IUF (Inverse User Frequency), DEF (Default voting) and DEFIUF (Default voting with IUF).

6.1 Neighbourhood Formation Time

The mean time to compute neighbourhoods is shown in Table 1. The Quit method is the fastest, since only a few inverted lists are examined. The Continue method is sometimes a little slower than normal inverted search. We implemented the accumulators for Quit and Continue as dynamic structures (hash tables) and when the stop criterion is as high as 10000 users, this structure has less of an advantage over arrays.

Timing results for inverted search and vector search for the Pearson correlation for one of the runs are shown in Figure 1 and Figure 2. The x axis shows the size of the user profile and the y axis the average number of milliseconds to compute a neighbourhood for that profile size.

The improvement in mean neighbourhood formation time by using inverted search is very encouraging. We see not only that inverted search is much faster on neighbourhood formation when given small profiles (up to a few hundred titles), but also for the very largest.

In general, inverted search is many times faster than in-memory vector search, even when using such a simple compression algorithm as variable byte coding. Furthermore, all data in the inverted structure resides on disk. By caching parts of the index and the inverted lists, it is of course possible to further improve the neighbourhood formation time.

6.2 Predictive Accuracy

The mean absolute error and rank score for the *AllBut1* protocol is listed in Table 2 and Table 3.

The Quit method did not always find neighbours to all

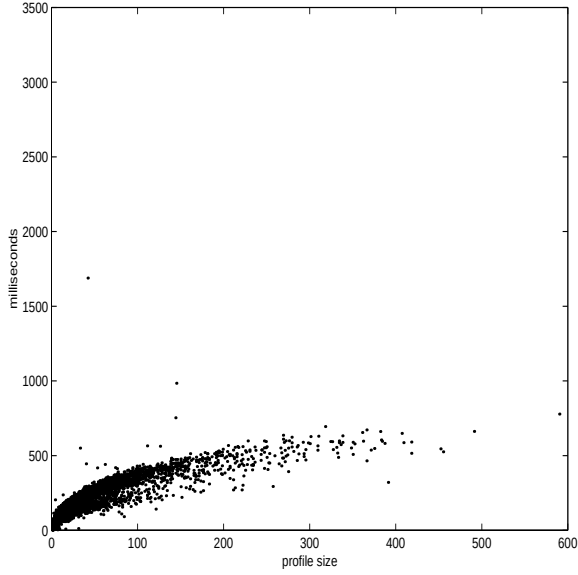


Figure 1: Inverted search time

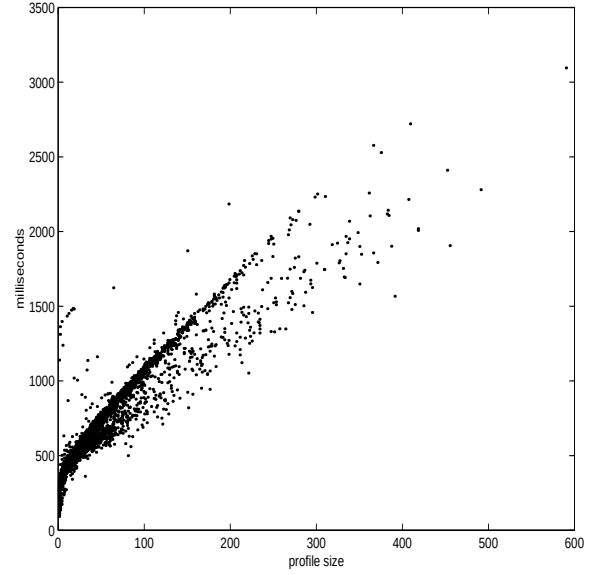


Figure 2: Vector search time

Table 2: Prediction results (MAE)

	CORR	IUF	DEF	DEFIUF
Algorithm				
VEC	1.08	1.07	1.03	1.05
INV	1.08	1.06	1.03	1.05
QUIT	1.03	1.02	1.04	1.03
CONT	1.06	1.05	1.00	1.03

users in the test set. On the CORR, IUF, DEF and DEFIUF it calculated neighbourhoods to 99%, 75%, 97% and 100% of the users, respectively. By contrast, the Continue strategy did find the same number of users as inverted search and vector, so the results among these variants are truly comparable.

The first interesting result is that the Continue strategy is the top performer in terms of rank score. Since we sorted the user profiles in descending weight order, this means that the shorter inverted lists (titles that have fewer votes) have high discriminating power, and that titles with a large number of votes actually corrupt the ranking.

We noted that Continue did not necessarily find exactly K neighbours, so another conclusion that we can draw is that this number should not be fixed, but dependent on which votes are in the user profile.

The second result is that the Quit method performs well on the users for which it could find correlations. We noted that in many cases only one inverted list was used by Quit. After all, if the size of the first list is greater than 10000, no more lists will be traversed. Furthermore, since Quit does not compute the true correlation (the accumulator sums are not complete), the effect of the pre-calculated values in the static arrays has yet to be analyzed. By varying the number of accumulators in Quit, and setting another threshold also on the number of inverted lists to traverse, we can find whether the method is useful.

In terms of mean absolute error, there is no large difference between any of the methods. Continue is still the top

Table 3: Prediction results (RS)

	CORR	IUF	DEF	DEFIUF
Algorithm				
VEC	7.37	13.13	24.42	18.04
INV	7.17	13.26	24.43	18.03
QUIT	17.94	16.37	21.61	18.14
CONT	10.65	11.40	27.50	20.78

performer, and performs equally well or better than normal inverted search. Quit performs well compared to not using termination heuristics at all. This means that if we are interested in single predictions, we could do well by only taking into account a few titles from the profile.

We found that default voting was the most effective correlation method, both in terms of mean absolute error and rank score. This is a little surprising, since the top performer in a previous study [3] was default voting with inverse user frequency. One explanation for this could be that our training set was much larger than the one used in the previous study.

7. CONCLUSIONS

Collaborative filtering is a relatively new technique for information filtering. It has been successfully used in domains where the information content is not easily parsable and traditional information filtering techniques are difficult to apply. We believe that it is useful as a complement to traditional content based filtering techniques.

There are however a number of problems with collaborative filtering that needs to be resolved before it is truly useful. One important problem is that of scalability, i.e. how to manage a large number of users and titles. Our way of attacking this problem is to cast the collaborative filtering problem into an information retrieval setting.

Our hypothesis was that by the use of inverted files, we could speed up execution time and perhaps even improve the predictive performance with early termination heuristics.

In order to test this hypothesis, we selected four well-known algorithms for collaborative filtering and expressed these in a form suitable for inverted retrieval. We selected a large test set for evaluation, and found some interesting results:

- The inverted file search algorithms were generally many times faster than in-memory search, even when having an equal amount of pre-calculated values.
- The Continue termination heuristic was the top performing algorithm in terms of rank score, and about as fast as normal inverted search.
- The Quit termination heuristic was very fast compared to the other methods tested, and yielded good predictions as well. However, in one case the method did not find neighbours to the same number of users as the other methods, and has to be further investigated.

Research is needed on strategies for updating the inverted files, and we hope that this paper will lead to further discussions on the subject of using inverted files for collaborative filtering.

8. ACKNOWLEDGEMENTS

The authors would like to thank Jussi Karlgren, Preben Hansen and the anonymous reviewers for useful comments on the paper.

9. REFERENCES

- [1] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Trans. on Database Systems*, 2(1):11–26, March 1977.
- [2] D. Billsus and M. J. Pazzani. Learning collaborative information filters. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 46–54. Morgan Kaufmann, 1998.
- [3] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, Madison, WI, July 1998. Morgan Kaufman Publisher.
- [4] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proceedings of the eight annual international ACM SIGIR conference on Research and development in information retrieval*, 1985.
- [5] M. J. Folk, B. Zoellick, and G. Riccardi. *File Structures: An Object-Oriented Approach with C++*. Addison-Wesley, 3rd edition, 1998.
- [6] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl. An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, 1999.
- [7] J. Karlgren. Stylistic experiments in information retrieval. In T. Strzalkowski, editor, *Natural Language Information Retrieval*. Kluwer, Dordrecht, 1999.
- [8] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl. Grouplens: Applying collaborative filtering to usenet news. *Communications of the ACM*, 40(3):77–87, 1997.
- [9] W. S. Lee. Collaborative learning and recommender systems. In *Proceedings of the Eighteenth International Conference on Machine Learning*. Morgan Kaufmann, 2001.
- [10] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
- [11] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Riedl. Application of dimensionality reduction in recommender system – a case study. In *ACM WebKDD 2000 Web Mining for E-Commerce Workshop*, 2000.
- [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 *Proceedings of the 1998 Conference on Computer Supported Cooperative Work*, Nov 1998.
- [13] U. Shardanand and P. Maes. Social information filtering: Algorithms for automating “word of mouth”. In *Proceedings of ACM CHI’95 Conference on Human Factors in Computing Systems*, volume 1, 1995.
- [14] H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.
- [15] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, 2nd edition, 1999.