

Web-Collaborative Filtering: Recommending Music by Spidering The Web

William W. Cohen
AT&T Shannon Laboratories
180 Park Ave Florham Park, NJ 07974
wcohen@research.att.com

Wei Fan
Department of Computer Science
Columbia University New York, NY 10027
wfan@cs.columbia.edu

Abstract. We show that it is possible to collect data that is useful for collaborative filtering (CF) using an autonomous Web spider. In CF, entities are recommended to a new user based on the stated preferences of other, similar users. We describe a CF spider that collects from the Web lists of semantically related entities. These lists can then be used by existing CF algorithms by encoding them as "pseudo-users". Importantly, the spider can collect useful data without pre-programmed knowledge about the format of particular pages or particular sites. Instead, the CF spider uses commercial Web-search engines to find pages likely to contain lists in the domain of interest, and then applies previously-proposed heuristics [Cohen, 1999] to extract lists from these pages. We show that data collected by this spider is nearly as effective for CF as data collected from real users, and more effective than data collected by two plausible hand-programmed spiders. In some cases, autonomously spidered data can also be combined with actual user data to improve performance.

Introduction

One key issue facing the field of computer science is how to exploit the vast amounts of knowledge available on the Web. The main problem, of course, is that the bulk of information on the Web is designed to be read by humans, not by machines.

To date, most programs that collect knowledge from the Web fall into two classes. The first class of knowledge-collection programs are the spiders employed by search engines (like Altavista) to index the Web. These spiders have limited understanding of the pages they collect, and hence only limited use (for instance, keyword search) can be made of the collected data. However, because the spiders are simple, and because they can access all sites equally well, large amounts of data can be easily collected. Below, we will call this the **oblivious spider** approach to data collection.

The second class of knowledge-collection programs are those employed by Web-based **information integration systems** (e.g., Whirl [Cohen, 1998], Ariadne [Knoblock et al, 1998], and others). These systems extract information from selected Web pages, and then store the information in an internal database. This approach allows more complex queries to be answered; however, to extract database-like information from a Web page requires either learning (e.g., [Kushmerick et al, 1997; Muslea et al, 1998]) or programming (e.g., [Hammer et al, 1997]) a special "wrapper" for that page. Because constructing wrappers is time-consuming, Web-based information integration systems tend to operate in limited domains.

Below we will call this data collection strategy the **programmed spider** approach. We use this term to emphasize the difference between this approach to data collection and the use of oblivious, search-engine like spiders, which do not need to be modified in any way to work on a new domain.

In this paper, we show that oblivious spidering can be used to collect data that is useful for the task of **collaborative filtering (CF)** [Hill et al 1995; Resnick et al 1994; Shardanand and Maes 1995; Soboroff et al 1999]. In collaborative filtering, entities are recommended to a new user based on the stated preferences of other, similar users. (For example, a CF system might suggest the band "The Beatles" to the user "Fred" after noticing that Fred's tastes are similar to Kumar's tastes, and that Kumar likes the Beatles.) More specifically, we will describe an oblivious spider that uses heuristics to collect lists of musical artists from the Web. These lists can be used to supplement or replace user ratings in a collaborative filtering system. Using actual user-log data, we measure the performance of several CF algorithms. We show that running a CF algorithm using data collected from an oblivious spider is nearly as effective as using data collected from real users, and better than using data collected by two plausible programmed spiders. In some situations, the oblivious-spider data can also be combined with user-log data to improve CF performance.

The work reported here complements previous research demonstrating that CF can be improved by using "content" features obtained from the Web by programmed spiders (e.g., [Basu et al, 1998; Good et al, 1999]). The main novel contribution of this paper is to show that CF can also be performed using data collected by oblivious spiders. The main technical innovations exploited in this paper are new, robust methods for extracting lists of entities from HTML pages [Cohen, 1999]. The list-extraction methods we use here were originally developed to facilitate building "wrappers" (that is, programmed spiders) for information integration systems.

A collaborative filtering problem

The dataset

We elected to explore the issues discussed above for a specific CF problem - the problem of recommending music. The primary dataset we used was drawn from user logs associated with a large (2800 album) repository of digital music, which has been made available for limited use within the AT&T intranet for experimental purposes. A server log records which files were downloaded by which IP addresses. Files on the repository obey certain naming conventions, which can (usually) be used to determine which musical artist - i.e., performer or composer - is associated with a downloaded file. (Some files are not associated with any single artist, but rather with some group or collection of artists. In the experiments reported below, downloads of these files were simply discarded.) Thus, by analyzing the log, it is easy to build up a record of which musical artists each client IP address likes to download.

We took 3 months worth of log data (June-August 1999), and split it into a baseline **training set** and a **test set** as follows. The test set consists of all downloads over the 3-month period associated with IP addresses that connected for the first time in August. The training set consists of all remaining downloads, i.e., downloads associated with IPs connecting in June or July; hence the train and test sets are disjoint. Recall that CF involves recommending **entities** (e.g., movies, books, etc) to **users**. In our experiments, we take musical artists to be entities, and assume that each client IP address corresponds to a different user. We constructed binary preference ratings by further assuming that a user U "likes" an artist A if and only if U has downloaded at least one file associated with A . We will denote the "rating" for artist A by user U as $rating(U, A)$: hence $rating(U, A)=1$ if IP address U has downloaded some file associated with A and $rating(U, A)=0$ otherwise. If $rating(U, A)=1$ then we say U has given A a **positive** rating; otherwise, we say U has given A a **negative** rating.

The baseline datasets are fairly large: there are 5,095 downloads from 353 IP addresses in the test set, 23,438 downloads from 1,028 IP addresses in the training set, and a total of 981 different artists associated with these downloads. It should be noted that in this dataset, almost all ratings (nearly 98%) are negative; thus, one would expect that more information about a user's preferences would be conveyed by a positive rating than by a negative rating.

It should be noted that the baseline training and test datasets are only an approximate reflection of real user preferences. One problem is the assumption that each IP address corresponds to a distinct user. In fact, while many IP addresses are static and correspond to a single-user workstation, some of the IP addresses are dynamic, and hence correspond to a session by some user, or worse, to a set of distinct sessions by different, unrelated users. Further, some users also access music from several fixed IP addresses (e.g., a home PC and a work PC), and conversely some fixed IP addresses might be used by several distinct users (e.g., a home PC that is used by several family members). Another issue is that many users only download a few files; in this case, it is certainly wrong to assume that all artists *not* downloaded are disliked.

However, although the data is noisy, it seems reasonable to believe metrics based on it can be used for comparative purposes. We note also that CF systems which can learn from this sort of noisy "observational" data (e.g., [Liebermann, 1995; Perkowit & Etzioni, 1997]) are potentially far more valuable than CF systems that requires explicit noise-free ratings.

Evaluation method

Many different evaluation metrics have been proposed for CF [Good et al, AAAI99]. In choosing an evaluation metric we found it helpful to assume a specific interface for the recommender. Currently, music files are typically downloaded from this server by a browser, and then played by a certain "helper" application. By default, the most popularly used helper-application "player" will play a file over and over, until the user downloads a new file. We propose to extend the player so that after it finishes playing a downloaded file, it calls a CF algorithm to obtain a recommended artist A , and then plays some song associated with artist A . If the user allows this song to play to the end, then this will be interpreted as a positive rating for artist A . Alternatively, the user could download some new file, overriding the recommendation. This will be interpreted as a negative rating for artist A .

For the sake of completeness, we will assume that the player can also be invoked without specifying the first song to be downloaded, leading to the following user interaction with the proposed "smart player":

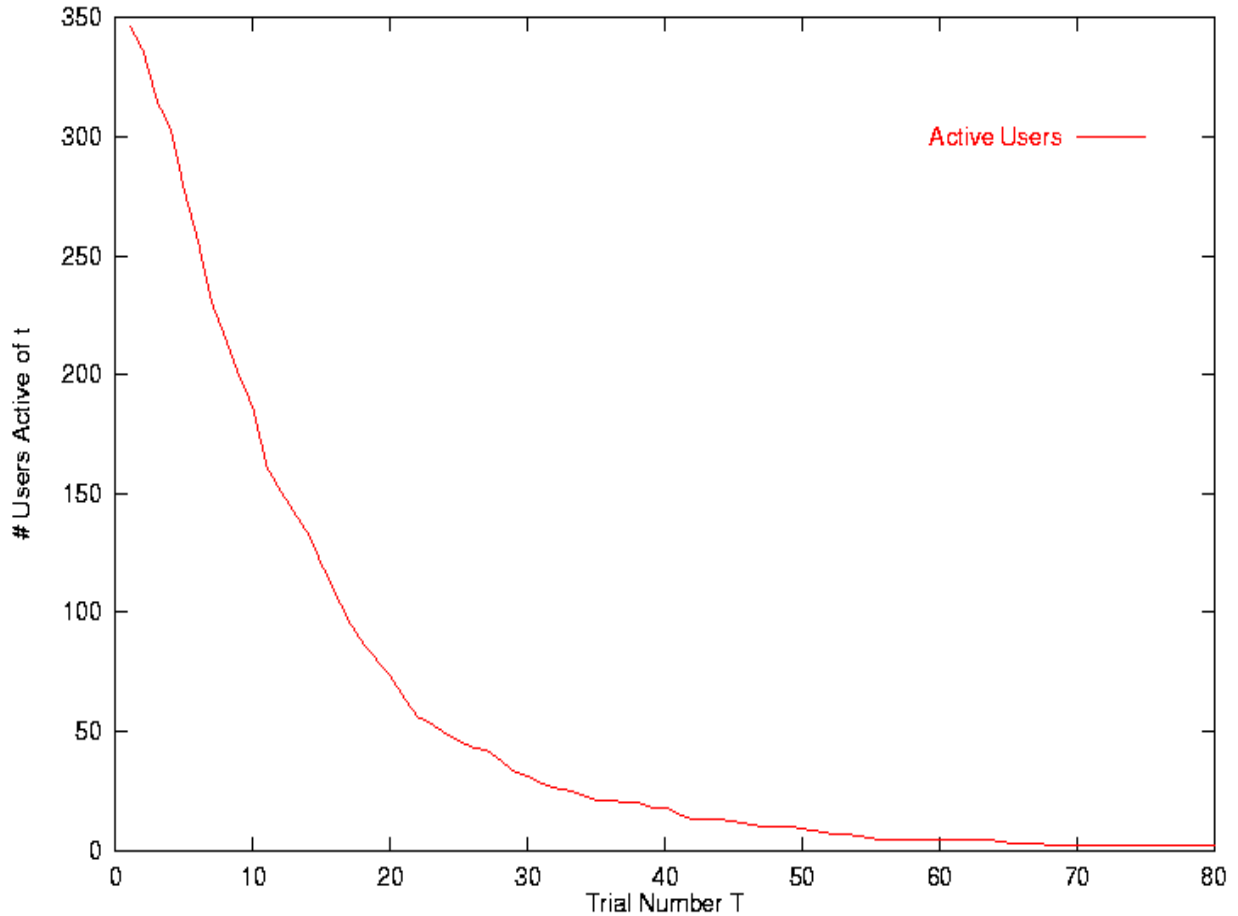
For $t=1, \dots, T$:

1. PLAYER: Call the CF algorithm to get a recommended artist, $A[t]$. Play some song by $A[t]$.
2. USER: Either
 - reject the song, by explicitly requesting another music file, or
 - accept the song (implicitly), by allowing it to play to completion.

This interaction can be simulated using the test user-log data. To simulate the user's actions, we accept a recommendation for A if A is rated positively by the test user, and reject it otherwise. When a recommendation is rejected, we simulate the user's choice of a new file by picking an arbitrary positively-rated artist. We continue the interaction until every artist rated positively by the test user has been recommended (or else is explicitly requested by the test user). We denote the total number of trials

T associated with the simulated interaction as $LEN(U)$.

Note that $LEN(U)$ is bounded by the number of artists rated positively by U . For most of the users, $LEN(U)$ is relatively small: the median value of $LEN(U)$ is only 10. This is shown in the figure below, which plots t against the total number of users U in the test set such that $LEN(U) \geq t$. We will typically truncate all of our result graphs at $t=50$, as beyond this point there are only a handful of distinct users.



The number of test-set users active at trial t

We define the **accuracy** of a simulated interaction between a recommendation method M and a test user U , denoted $ACC(M, U)$, to be the number of times the user accepts a recommendation, divided by $LEN(U)$. For instance, if 10 recommendations were made, and three were accepted, then accuracy would be 30%. We define $ACC(M, U, t)$ to be the accuracy over the first t trials. To simplify notation, we will drop the first argument and write $ACC(U)$ or $ACC(U, t)$ when M is clear from context.

Baseline experimental results

Algorithms used

The CF algorithms we used all have the following behaviour. For each trial t , the algorithm is called with two inputs:

1. a list of artists $A[1], \dots, A[t-1]$ that have been explicitly rated by some user U ; and
2. the ratings for these artists: $\text{rating}(U, A[1]), \dots, \text{rating}(U, A[t-1])$.

The algorithm then associates a numeric **score** with each artist A . A higher score is interpreted as stronger recommendation for A . Finally, a specific artist $A[t]$ is recommended by picking the highest-scoring A that does not appear in the list of previously-rated artists, $A[1], \dots, A[t-1]$. Below we will denote the score of A by $\text{SCORE}(A)$.

The algorithms used were the following.

- **Popularity (POP):** This algorithm simply scores each artist according to the total number of users in the training set that rate the artist favorably. That is, if $U[1], \dots, U[n]$ are all the users in the training set, then

$$\text{SCORE}(A) = \text{rating}(U[1]) + \dots + \text{rating}(U[n])$$

This algorithm is that it is perhaps the simplest plausible recommendation scheme that depends on the training set; it is analogous to using the "Billboard Top-40" to pick music or the New York Times best-seller list to pick books. Although it predates any academic research on CF, it is often rather hard to beat according to objective standards of performance.

- **K-nearest neighbour (K-NN):** This algorithm first finds the k users $U[1], \dots, U[k]$ whose preferences are most similar to U . Specifically, we define the **distance** between U and U' relative to the set of artists $A[1], \dots, A[t-1]$ to be:

$$\text{DIST}(U, U') = |\text{rating}(U, A[1]) - \text{rating}(U', A[1])| + \dots + |\text{rating}(U, A[t-1]) - \text{rating}(U', A[t-1])|$$

The K-NN algorithm first finds the k users $U[1], \dots, U[k]$ in the training set that are closest to U according to this metric. Ties are broken by preferring neighbors $U[i]$ with fewer positive ratings. (Recall that most ratings are negative; hence agreement on positive ratings is less likely *a priori* than agreement on negative ratings.) Artists are then scored according to their popularity in the set of neighbouring users:

$$\text{SCORE}(A) = \text{rating}(U[1], A) + \dots + \text{rating}(U[k], A)$$

The K-NN method is one of the most widely-used CF algorithms (e.g., [Hill et al 1995; Shardanand and Maes 1995]). In our experiments we used $k=10$.

- **Weighted majority (WM):** This is a slight modification of a recommendation algorithm suggested by Nakamura and Abe [1998]. This algorithm has strong formal foundations, and can be implemented quite efficiently (at least in principle). It is based on computing "weights" for a large number of very simple recommendation strategies called "experts", each of which predicts ratings

for some subset of artists and users. Let r be a learning rate, where $0 < r < 1$. If an expert E has made p correct predictions and n incorrect predictions in the past, then the **weight** of E , denoted $WT(E)$, is defined as

$$WT(E) = (2-r)^p * r^n$$

Now fix an artist A and a test user U , and let $E[1], \dots, E[m]$ be all the experts that predict $rating(U, A) = 1$, and let $F[1], \dots, F[n]$ be all the experts that predict $rating(U, A) = 0$. Then the score of A is defined to be the difference of the total weight on the "positive" (E) experts and the total weight on the "negative" (F) experts:

$$SCORE(A) = WT(E[1]) + \dots + WT(E[m]) - WT(F[1]) - \dots - WT(F[n])$$

In our implementation we used $r = 0.5$, and these experts:

1. For each pair of artists $A[i]$ and $A[i']$, we define a "expert" $E[i, i']$ which predicts that all users will rate $A[i]$ and $A[i']$ identically.
2. For each pair of users $U[j]$ and $U[j']$, we define a "expert" $E[j, j']$ which predicts that $U[j]$ and $U[j']$ will rate all artists identically.
3. For each artist $A[i]$, we define an expert $E[i]$ which predicts that all users will rate $A[i]$ positively.

The first two sets of experts follow Nakamura and Abe [1998], and the final set of experts was added to improve performance on the first few trials for a user. (Without these experts, performance is no better than random guessing when $t=1$.) We also found that performance was somewhat better when the number of correct predictions p assigned to an expert did not include cases in which a negative training-set rating was correctly predicted on the basis of another negative training-set rating. (Again, since most ratings are negative, this sort of agreement is quite likely to happen by random chance.) In recommendation, experts are trained on all ratings from the training set, as well as the known preferences from the test user U .

- **Extended direct Bayesian prediction (XDB):** This algorithm was motivated by the observation that, under relatively weak statistical assumptions, it is relatively easy to recommend optimally (or nearly optimally) in the first two trials of each interaction. For instance, on trial $t=1$, the optimal prediction is clearly the one made by POP, if one assumes that users are i.i.d. For trial $t=2$, the situation is only slightly more complicated. Here we have one positive rating by the test user. Depending on the outcome of the first trial, we may or may not have one negative rating: however, since negative ratings carry little information, in either case we will ignore this. Now consider what the optimal behavior is given a single positive rating, i.e., a single artist $A[i]$ that user U is known to like. The probability that U will like artist $A[j]$ is simply

$$\text{Prob}(rating(U', A[j])=1 \mid rating(U', A[i])=1)$$

where the probability is taken over possible users U' . This probability can be easily estimated from the training data. In the experiments, we use an **m-estimate** [Mitchell, 1997] for this probability, where $m=1$ and the prior is simply the prior probability of a positive rating. Let $R(A[j], A[i])$ denote this estimate. (Informally, $R(*, *)$ measures the "relatedness" of two artists.) For trial $t=2$, it is natural to use the scoring function $SCORE(A[j]) = R(A[j], A[i])$ in making a recommendation.

Unfortunately, this approach cannot be used on later trials, because it becomes too difficult to estimate the necessary statistics. Thus, we experimented with a simple *ad hoc* extension of this

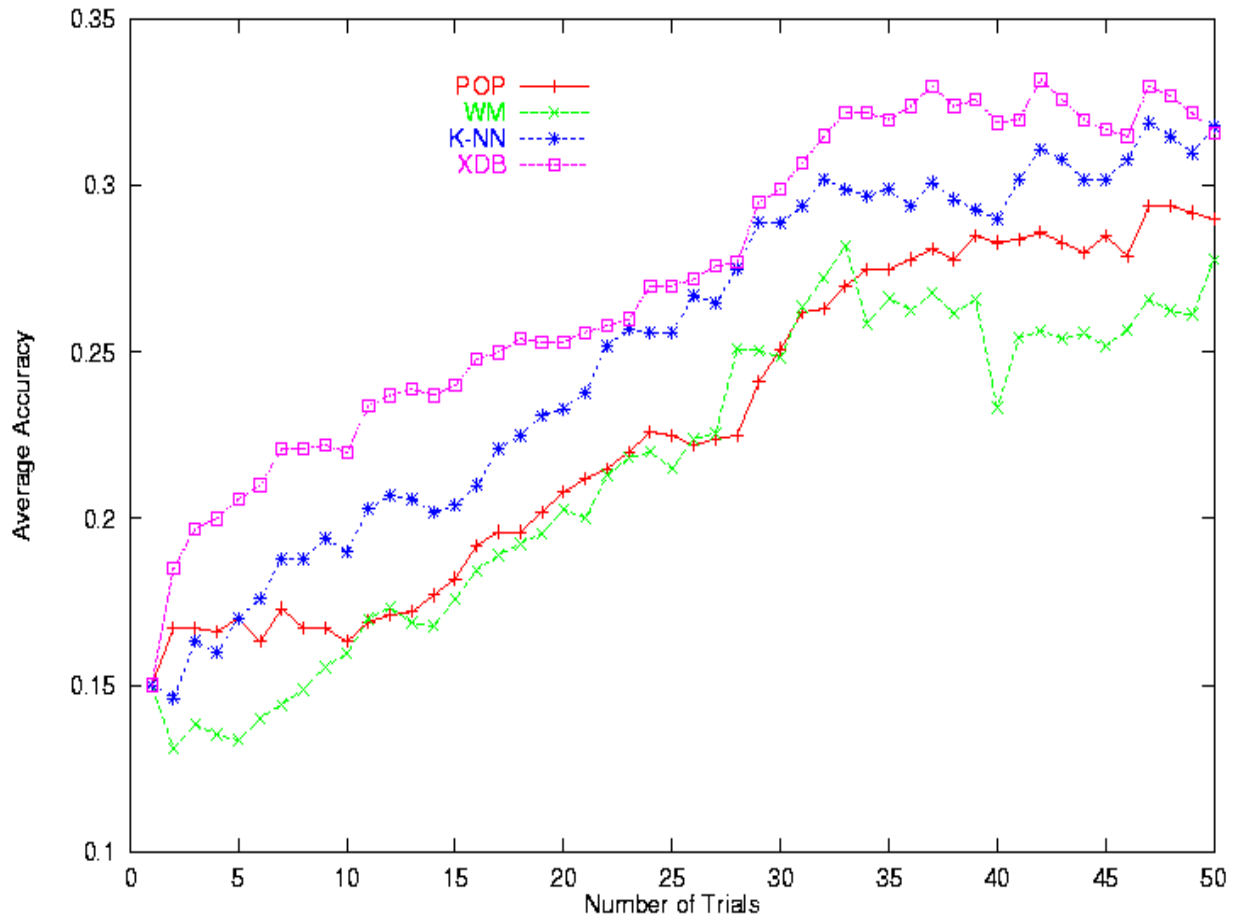
"direct Bayesian" recommendation scheme. Consider an arbitrary trial t , and let $L[1], \dots, L[n]$ be the artists that have been positively rated by U . The scoring system used by XDB for $t=1$ is the same as that used by POP, and for $t>1$ the scoring function is

$$\text{SCORE}(A) = 1 - (1 - R(A, L[1])) * \dots * (1 - R(A, L[n]))$$

Informally, $\text{SCORE}(A)$ can be understood as $1 - \text{PENALTY}(A)$, where $\text{PENALTY}(A)$ is the probability that A is *not* "related" to any $L[i]$, assuming independence of the $L[i]$'s. Notice that $\text{SCORE}(A)$ equals $R(A, L[1])$ when $n=1$. Empirically, XDB works quite well, particularly when t is small.

Collaborative filtering with user data

Before describing how oblivious spidering can be used to collect training data for CF algorithms, we will first evaluate these algorithms by training them on the baseline training data derived from the server logs, and then testing them on the baseline test data. The figure below plots a trial number t on the x axis, and records on the y axis the average value of $\text{ACC}(U, t)$ over all U such that $\text{LEN}(U) \geq t$.



Baseline results for four CF methods using server-log data

To summarize, all of the methods far outperform random guessing (which has an accuracy of about 0.02 on this data) and appear to give usefully accurate recommendations. XDB and K-NN perform best, while WM's performance is generally at best comparable to POP.

In order to determine if the difference between two CF methods M_1 and M_2 is statistically significant, we used a paired t-test to test the null hypothesis that the expected values of $ACC(M_1, U)$ and $ACC(M_2, U)$ are the same. According to this test, XDB and K-NN are both statistically significantly better than POP. However, XDB is not statistically significantly better than K-NN. Further experiments will focus on the behavior of XDB and K-NN.

In passing, we note that these curves are *not* conventional learning curves, in which prediction performance on a single problem is measured as a function of the amount of training data; in these curves, performance is averaged over a *different* set of users at each point t . This effect explains, for

example, the upward-sloping shape of the results for POP, a non-adaptive recommendation strategy: clearly, it is easier to correctly guess preferred musical artists for users with many positive ratings than for users with few positive ratings.

Collaborative filtering with programmed spiders

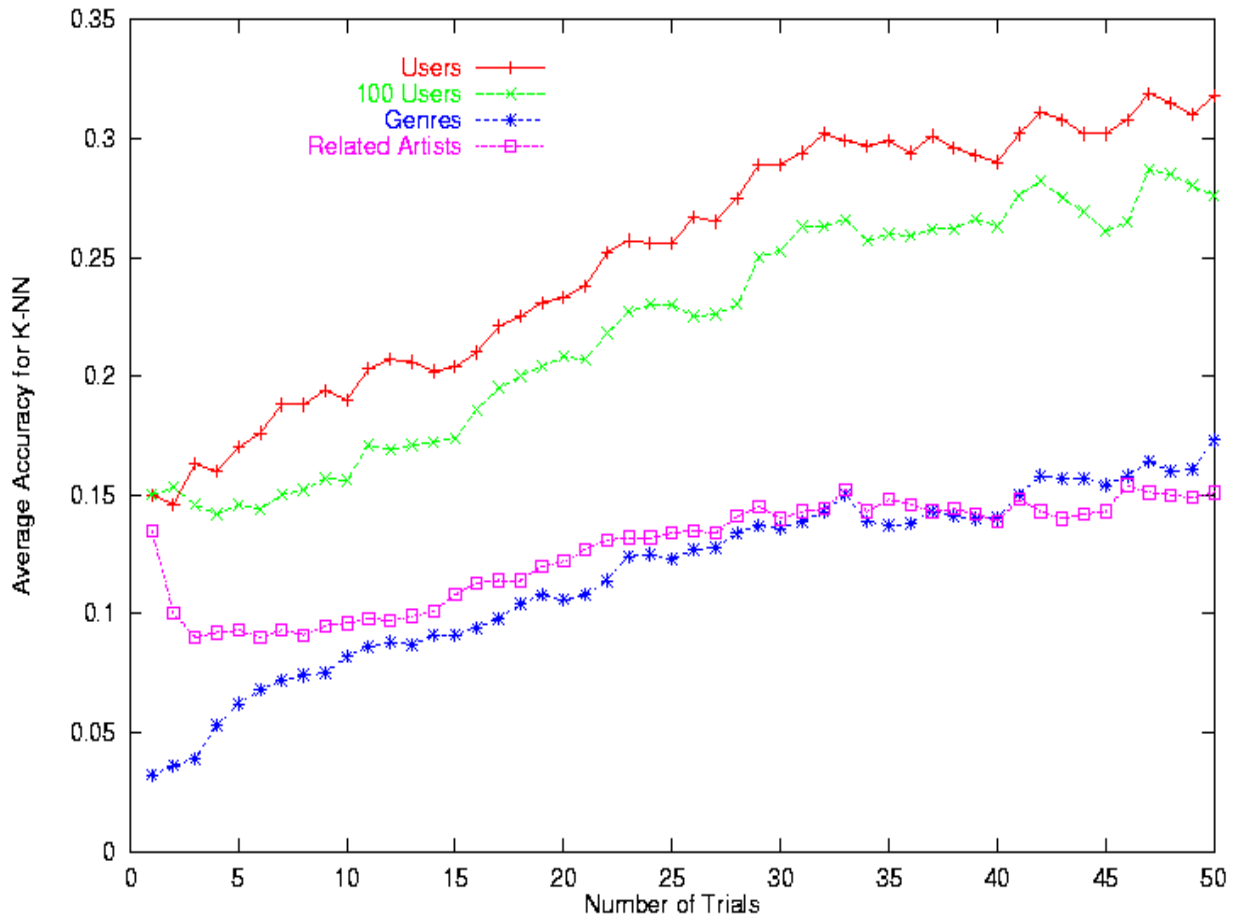
As an additional baseline, we also implemented two recommendation systems based on data collected from programmed spiders. We wrote spiders that traversed a large on-line music database (Allmusic.com) and extracted information about the "genre" and "musical style" of each artist. Genres are broad categories of music, like "jazz" or "country", and styles are narrow groupings of artists, like "country boogie". For each artist A , a list of "related artists" was also extracted. Our spider used a relatively broad definition of "related"; every artist that is mentioned in the on-line description of A is considered to be "related to" A .

One plausible way to use genre or style information is to find genres that correlate with the user's preferences, and then recommend artists in these genres. We implemented a genre-based recommendation system using the K-NN algorithm described above. For each genre, we constructed a **pseudo-user** that rates everything in the genre positively, and everything not in the genre negatively. We then used the standard K-NN algorithm described above, trained on these pseudo-users rather than the baseline training set, to make recommendations.

Used in this way, K-NN first finds genres (pseudo-users) that are correlated with the test user's positive ratings, and then recommends artists that play in these genres. The heuristics described above lead to a preference for relying on narrower groupings of artists, rather than broader categories (i.e., a preference for styles to genres), and for recommending artists that play in more than one correlated genre.

A similar approach was used to build a recommender that uses the related-artist information. In this case, each pseudo-user gives a positive rating to some artist A plus all artists "related to" A . The K-NN algorithm was again used for recommendation. Thus our related-artist recommendation system is much like a genre-based recommendation system, where each set of related artists acts like a (very narrow) genre.

The results for these systems are shown in the figure below, which compares the genre-based recommender, the related-artist recommender, and the baseline K-NN recommender (trained on the user data). We also show results for K-NN trained on a subset of the baseline User dataset including only 100 distinct users. The 100-User subset is approximately equal in size to the related-artist and genre datasets (see the table below). To summarize, the two programmed-spider recommendation systems are comparable in performance, with the related-artist system perhaps making slightly better recommendations when ϵ is small. However, performance with the programmed-spider datasets is statistically significantly worse than with either the User or 100-User datasets.



Results for the K-NN algorithm using data collected by a programmed spider

Similar results are (not shown) obtained if K-NN is replaced with the XDB or WM algorithm. We also note that there is relatively little difference between the baseline training set, with 1000-odd users, and the 100-user training set. This shows that the learning curve for CF in this domain is relatively flat: i.e., that little is gained by increasing the size of the training set.

An oblivious spider that collects CF data

In the experiments above, each training set could be represented as a set of users (or pseudo-users), together with a list, for each user, of the set of artists rated positively by that user. In the baseline experiments, these artist-lists were obtained by analyzing log data. Collecting this sort of data requires the substantial effort of serving a substantial user community for a long period of time. It is natural to ask if there are any other ways in which sets of "favorite artists" might be obtained, other than by examining user logs.

One possibility would be to conduct a series of Web searches with the goal of finding Web pages entitled something like "My favorite musical artists", and then extracting the list of artists in that page by some automated means. Further reflection suggests that other pages which contain lists of musical artists could at least plausibly be used to train a CF algorithm like K-NN. As examples, a user interested in country blues might find her preferences are well correlated with a page titled "Musical Influences of Robert Johnson"; a user interested in New Age music might find her preferences are correlated with a page titled "Top-selling New-Age Albums for the Week of September 13".

We thus elected to build an oblivious spider that simply looks for lists of musical artists, or more generally, lists of items of some known type. The list-extraction part of our spider is based on heuristics for list-extraction introduced by Cohen [1999]. The remaining portions rely on exploiting commercial Web-search engines to find pages likely to contain lists.

Extracting lists from a set of Web pages

We will first describe the heuristics for extracting lists from a set of Web pages. The sample page below will be used as a running example.

```
<head>...</head>
<body>
<h1>Biff's Favorite Bands</h1>
<ul>
<li><b>Alice in Chains</b> - completely awesome!
<li><b>Smashing Pumpkins</b> - these d00des rock!!!
<li><b>Barry Manilow (NEW)</b> - check out all these
        great <a href="biffmp3.htm">MP3's</a>!!!
...
</ul>
</body>
```

Part of an HTML page containing a list of musical artists

Our routine for list extraction, henceforth `FindLists`, takes as input a set N of entity names and a set $URL[1], \dots, URL[n]$ of URLs, and operates as follows.

Program `FindLists(N, URL[1], \dots, URL[n])`:

1. Download each $URL[i]$, and parse the HTML markup of the downloaded Web page.
2. For each HTML-tree node n that contains no more than 250 characters of non-markup text below it, construct a pair of the form $(Pos(n), Text(n))$, where $Text(n)$ is all non-markup text that appears below node n , and $Pos(n)$ is the concatenation of the URL for the page and the sequence of HTML tags that appear on the path from the root of the HTML-tree to n . The intuition here is that $Text(n)$ is a candidate entity name (e.g., the name of some musical artist), and $Pos(n)$ identifies a list of objects.

For instance, consider the sample HTML page above. The pairs extracted in this stage might include the following:

(`http://users.com/~biff+body.h1`, "Biff's Favorite Bands")

(http://users.com/~biff+body.ul.li, "Alice in Chains - completely awesome!")
(http://users.com/~biff+body.ul.li.b,"Alice in Chains")
(http://users.com/~biff+body.ul.li, "Smashing Pumpkins - these d00des rock!!!")
(http://users.com/~biff+body.ul.li.b,"Smashing Pumpkins")

...

3. From the set S of $(Pos, Text)$ pairs computed above, and the set N of entity names, construct all tuples of the form $(Pos, Text, A)$ such that $(Pos, Text)$ is a tuple in S , A is an entity name in N , and the **cosine similarity** between $Text$ and A is at least 0.8.

Cosine distance is a measure of textual similarity that is widely used in information retrieval. We use the variant of cosine distance implemented in the WHIRL system [Cohen, 1998], which can be conveniently used to perform the tuple-construction operation defined above.

For the sample HTML page above, the output of this stage would include the following triples:

(http://users.com/~biff+body.ul.li.b,"Alice in Chains", "Alice in Chains")
(http://users.com/~biff+body.ul.li.b,"Smashing Pumpkins", "The Smashing Pumpkins")
(http://users.com/~biff+body.ul.li.b,"Barry Manilow (NEW)", "Barry Manilow")

...

4. Note that the tuples are now very close to being in the desired format, namely, entity names paired with lists to which they belong. The final step of `FindLists` is to filter the list of triples constructed above by removing all triples associated with positions that appear less than 4 times, and discarding the second component of each triple. The resulting set of position-entity pairs is the output of `FindLists`.

For the sample HTML page above, the final output of this stage would be the pairs:

(http://users.com/~biff+body.ul.li.b,"Alice in Chains")
(http://users.com/~biff+body.ul.li.b,"The Smashing Pumpkins")
(http://users.com/~biff+body.ul.li.b,"Barry Manilow")

...

This output can be used as the input of a CF algorithm. Specifically, each position P represents a list of entities, and hence can be represented as a pseudo-user, where the pseudo-user for P rates positively exactly those entities that appear on the list P (that is, the set of entities A that appear in a pair with the position P).

While our example for `FindLists` is a single page, the implementation operates on a set of URLs, as some operations can be performed better if a large number of pages are processed in parallel. Experiments in a somewhat different context suggest that this sort of list extraction (based on HTML parse-tree positions and similarity to known entity names) is reasonably accurate [Cohen, 1999]. For the purpose of CF, however, it is probably the case that a good deal of inaccuracy in the extraction step can be tolerated, simply because CF algorithms are tolerant of noise.

Design of the spider

The CF spider uses the `FindLists` routine to extract lists from Web pages identified using certain simple heuristics. The spider has two phases. **Phase 1** constructs an initial set of entity-lists. This set of lists can be used in CF, but tends to have statistics very different from the user data described above, which limits its utility in CF. **Phase 2** uses the Phase 1 entity-lists to build a second group of entity-lists with more representative statistics.

The implementation of the CF spider currently consists of a set of Perl and shell scripts that must be manually invoked by a user in a specified order, and manually monitored for difficulties like transient network failure and so on. However, complete automation of the CF spider is technically straightforward.

In **Phase 1**, the CF spider performs these actions.

1. For each artist A , a Web search engine is invoked with the artist's name, and the top 100 URLs returns are recorded. In the music domain, this resulted in a list of 85,362 candidate URLs (using Altavista as the search engine).
2. The candidate URLs are filtered to find URLs that appear at least twice. (The assumption here is that these URLs are more likely to be lists.) In the music domain, there were 5,268 duplicated URLs.
3. The duplicate URLs are then passed to `FindLists`. In the music domain, this produced 2,519 pseudo-users with a total of 13,527 positive ratings (i.e., position-entity pairs).

The entity-lists constructed by `FindLists` could be used directly in CF. Unfortunately, these lists tend to have somewhat unnatural statistics: in particular, due to the systematic nature of the search on artist names, obscure artists tend to be over-represented, and popular artists are under-represented, relative to the user data on which we intend to test the system. **Phase 2** is intended to construct a more representative set of pseudo-users.

In **Phase 2**, the CF spider performs these actions.

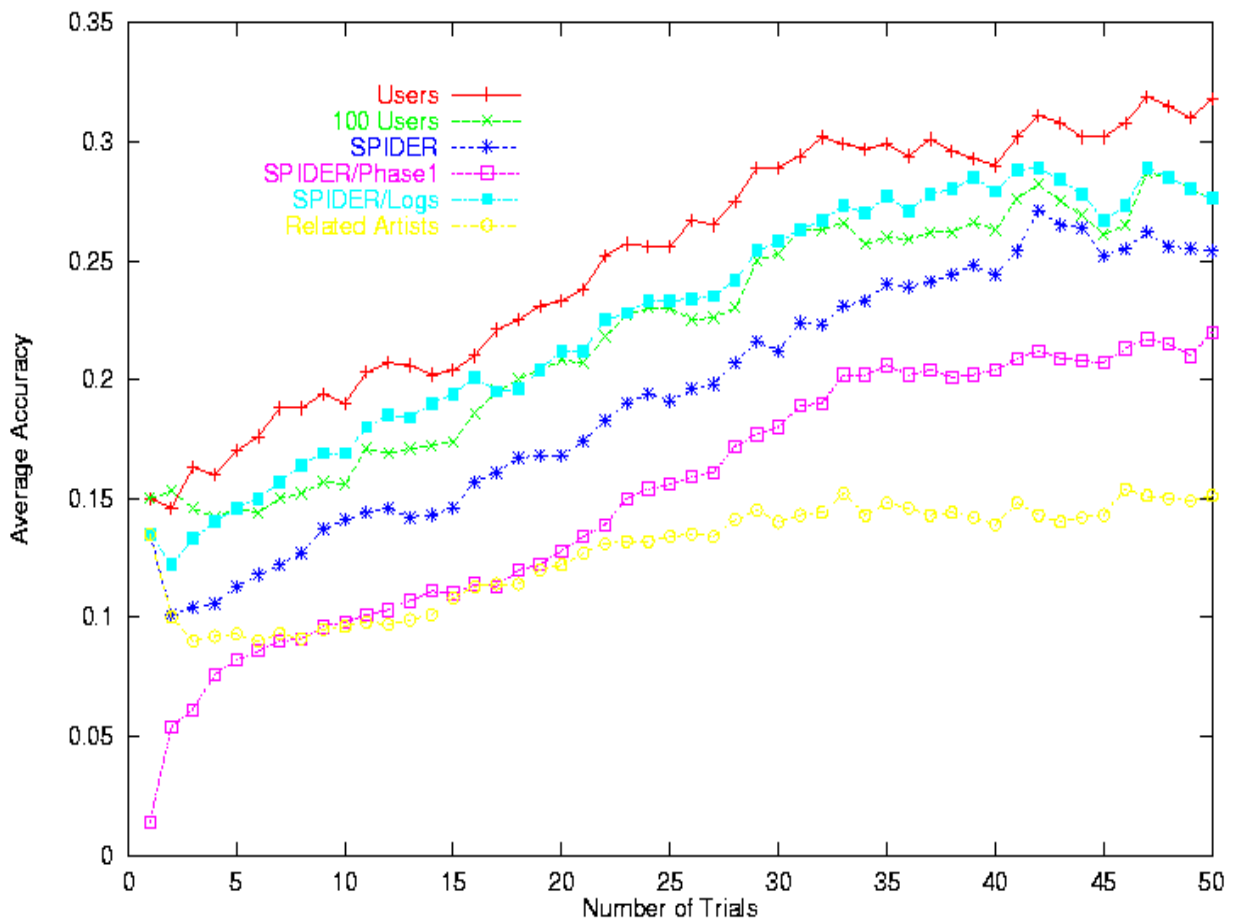
1. The CF spider finds the 1000 pairs of entity names that co-occur most frequently in the entity-lists constructed in Phase 1. This step was suggested by the observation that searching for a pair of artist names is likely to produce Web pages that are lists; furthermore, the co-occurrence statistics for the Phase 1 entity-lists are not too different from the co-occurrence statistics for the user data.
2. The first few entries from this list of pairs are manually inspected, and pairs consisting of different artists with highly similar names (according to cosine distance) were removed. (Such pairs are an unfortunate by-product of the `FindLists` method.) In the music domain, four pairs were removed. We conjecture that results would be similar if this step were not performed.
3. For each such pair $A[i], A[j]$, another Web-search was performed on this *pair* of names, in this case retrieving the top 10 URLs. (We used Northern Lights as the search engine here, resulting 4,326 unique downloadable URLs in the music domain.
4. Finally, these URLs were processed with `FindLists`. This produce 1,894 pseudo-users, with 48,878 positive ratings.

Results using obviously-collected Web data

The figure below plots the performance of the K-NN algorithm using obviously-collected datasets. For comparison, we also plot the performance of K-NN on the baseline User training set, the 100-User

subset of the baseline training set, and the related-artist dataset. The results for K-NN and the output of the oblivious CF spider are shown on the line labeled SPIDER. To summarize, SPIDER performs significantly better than the related-artist dataset, but significantly worse than the User dataset. SPIDER is statistically indistinguishable from the 100-User dataset.

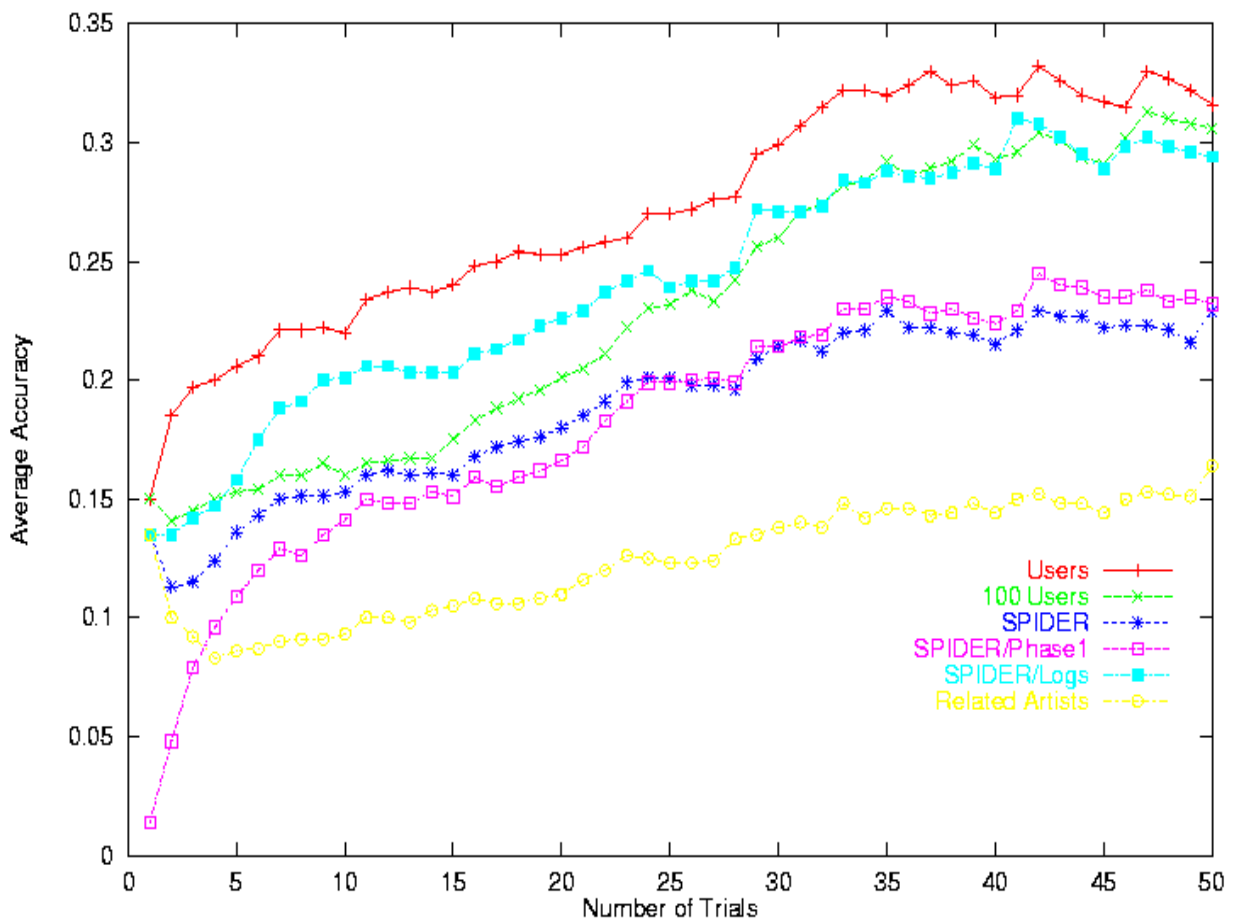
We also evaluated two other obviously-collected datasets. SPIDER/Phase1 is the output of Phase 1 of the spider. Results on this dataset are significantly worse than with SPIDER, demonstrating that Phase 2 is useful. SPIDER/Logs is the output of running Phase 2, starting with the baseline User training set instead of the output of Phase 1. Unlike the SPIDER dataset, the SPIDER/Log dataset cannot actually be constructed without having access to user logs, so it is not a viable replacement for this log data; however, it is nonetheless informative to consider its performance. Performance with SPIDER/Logs is significantly better than with the SPIDER dataset, even though it is smaller (26,917 pairs with 1,539 pseudo-users). This may indicate that even after Phase 2, the statistics of the data are still somewhat skewed relative to the user data. (We note that users of the music server are primarily employees of AT&T Research, a rather atypical population.)



Performance of the K-NN algorithm with User data, compared to data collected by an oblivious spider.

Qualitatively similar results were obtained with the XDB, WM and POP algorithms. The results for XDB are shown in the figure below. In this case the SPIDER dataset is significantly better than the genre and related-artist datasets, but significantly worse than both the User and 100-User datasets.

XDM with the SPIDER/Logs dataset also performs quite well - in this case, significantly better than with the 100-User dataset. XDM with the SPIDER dataset dataset is again somewhat better than K-NN with the SPIDER dataset; however, the difference is not significant.



Performance of the XDB algorithm with User data, compared to data collected by an oblivious spider.

Dataset	# Users	# Pos. Ratings
User (baseline)	1,028	21,977
100 User	100	4,315
SPIDER/Log	1,539	26,917
SPIDER	1,894	48,878
SPIDER/Large	2,333	63,815
SPIDER/Phase1	2,519	13,527
Related Artist	639	5,104
Genre	613	4,623

Sizes of the various datasets

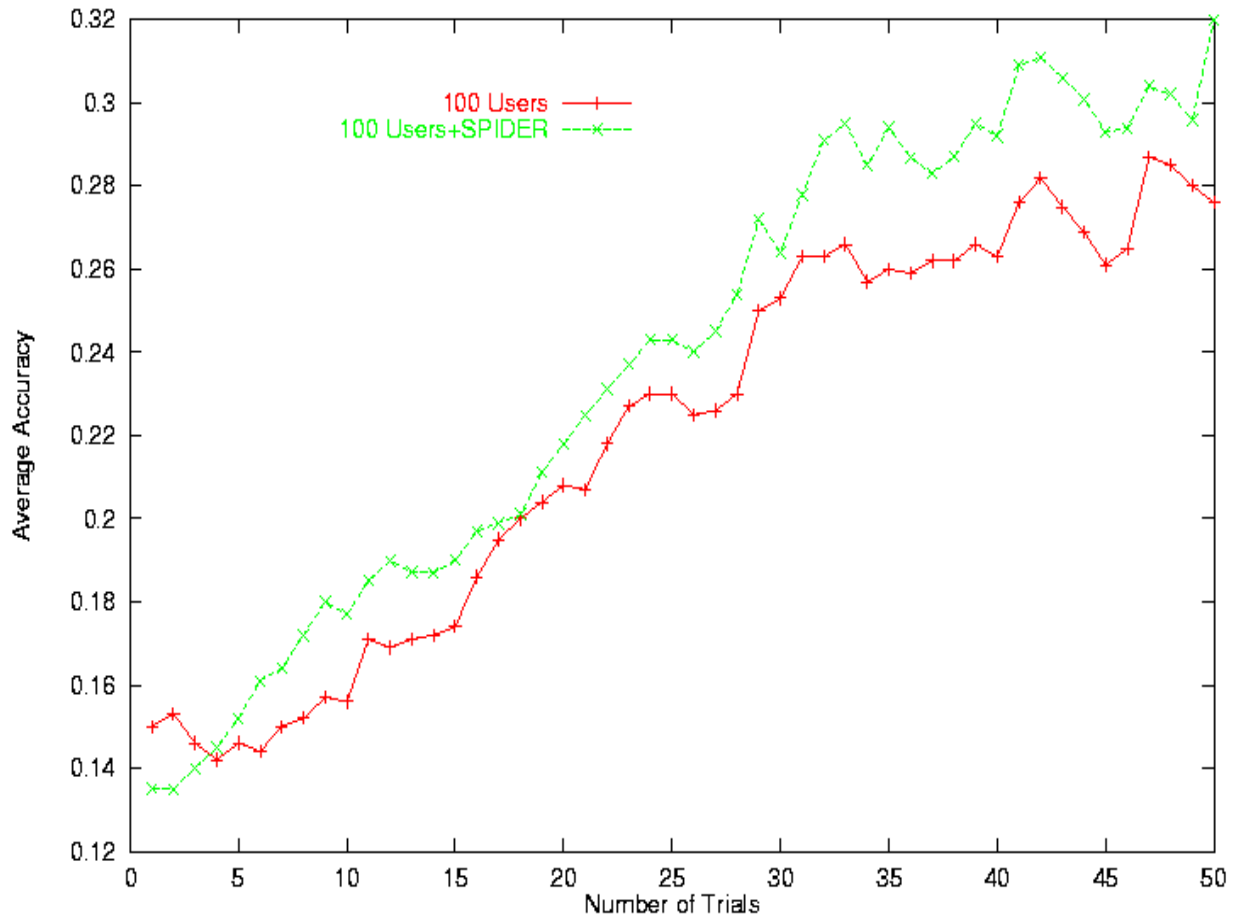
The table above summarizes the size of the datasets explored thus far. Here the datasets are sorted by performance. We note that it is hard to construct a natural dataset of predetermined size, due to the wide variability in the size and quality of web pages unearthed by the SPIDER, and also wide variability in $|\text{LEN}(U)|$. However, as previously noted, the learning curve appears to be fairly flat for this problem: e.g., the average difference in $\text{ACC}(u)$ between the User and 100-User datasets is only about 2.5%. We thus conjecture that the primary differences between datasets is based on the nature of the information they contain, not their sizes. Note that the best-performing SPIDER datasets are comparable in size to the User dataset.

As an additional test of this conjecture, we built a larger SPIDER dataset (SPIDER/Large in the table) by repeating Phase 2 of the CF spider, using Google as the search engine, and collecting the top 30 URLs for each artist pair. (In building SPIDER we collected the top 10 URLs.) Performance with SPIDER/Large is not significantly better than with SPIDER for any of the K-NN, XDB, or POP algorithms (in fact performance is slightly worse for two of the three algorithms) confirming the flatness of the learning curve.

Combining user data and obviously-collected data

We have thus far considered *replacing* user data for a CF algorithm with data collected by spiders. We will now consider *augmenting* user data with data collected by spiders, where "augmenting" simply means adding to the first dataset all the pseudo-users from the second dataset. In augmenting user data with obviously-collected data, we primarily used the Spider/Log dataset, which was constructed by running Phase 2 of the CF spider with the user log data as input.

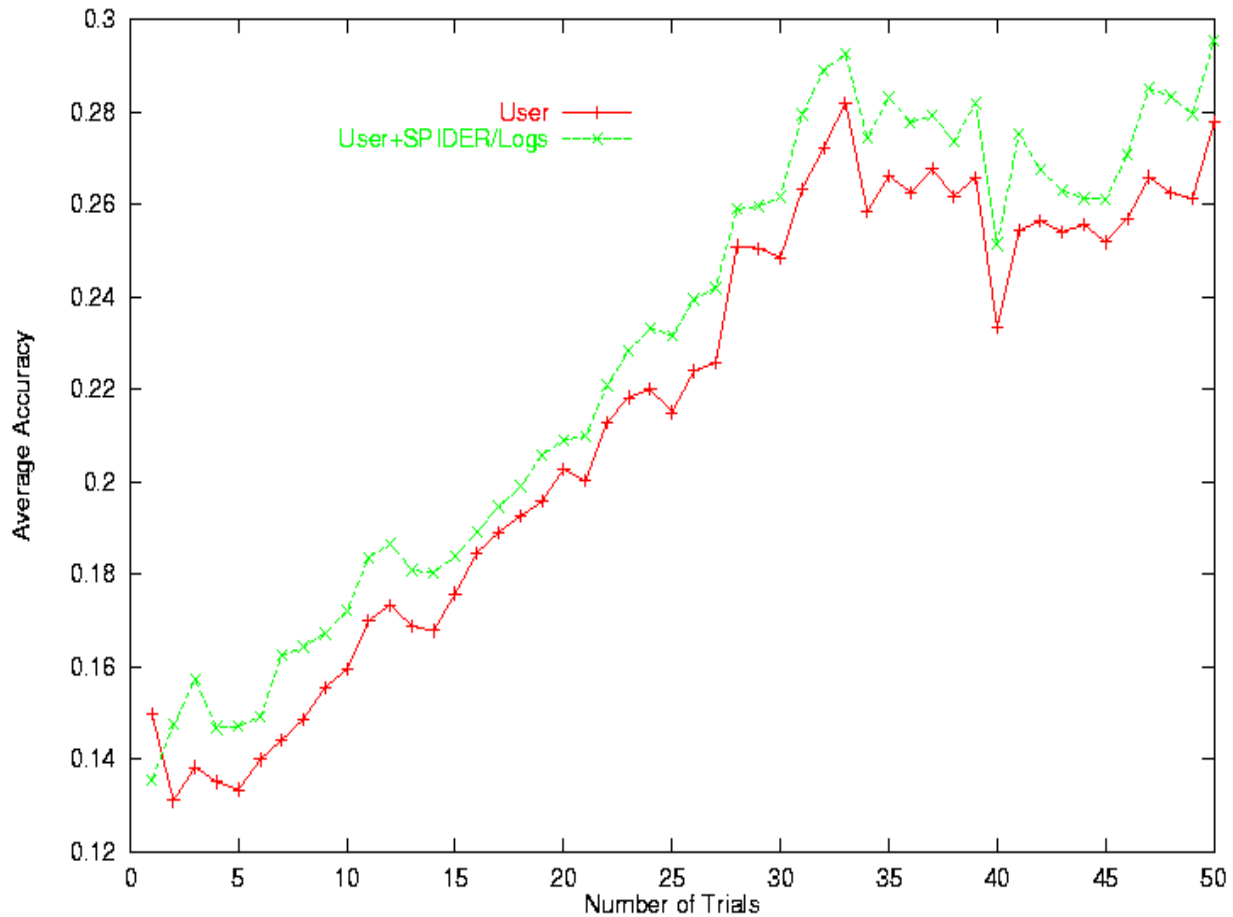
Our results thus far are preliminary, but intriguing. The figure below shows the result of combining the smaller 100-User training set with the Spider/Log dataset, and then using the K-NN algorithm. The augmented dataset shows a statistically significant improvement over the 100-User baseline.



Performance of the K-NN algorithm using the 100-User dataset augmented with obviously-collected Web data

Another statistically significant improvement (not shown) was gained by repeating this experiment with the XDB algorithm. Performance of the XDB algorithm was also significantly improved by augmenting the 100-User dataset with the related-artist dataset, or by the genre dataset. Interestingly, however, K-NN's performance did *not* improve by combining the 100-User dataset with either the related-artist dataset or the genre dataset. This suggests that obviously-collected data is in some settings more valuable than "content" features.

Unfortunately, augmenting the complete User dataset with the SPIDER/Log data did not improve performance for either K-NN or XDB: in both cases, performance on the combined dataset is statistically indistinguishable from performance on the User dataset alone. However, the WM algorithm *is* statistically significantly improved by augmenting the User baseline training set with the Spider/Log dataset, as can be seen in the figure below.



Performance of the WM algorithm using user data augmented with obviously-collected Web data

Conclusions

We have considered replacing or augmenting user data for a collaborative filtering (CF) algorithm with data collected by spiders from the Web. This is a special case of what we believe to be an extremely important research problem: namely, using automated methods to effectively exploit the vast amounts of information available on the Web.

Previous work has shown that CF can be improved by combining user data with "content" features obtained from the Web by special-purpose "programmed" spiders (e.g., [Basu et al, 1998; Good et al, 1999]). In our experiments we have shown that CF algorithms can also exploit data collected by an "oblivious" spider. In fact, in our experiments, the obviously-collected data leads to substantially better recommendation performance than the programmatically-collected data. This result holds even though the "programmed" spider uses extensive knowledge to extract data from a high-quality site filled with

expert recommendations; even though the oblivious spider relies on very simple heuristics to extract information from Web pages; and even though the number of pages from which information was actually extracted by the oblivious spider is rather small. (In the experiments, data from less than three thousand Web pages was actually used by the CF algorithms.) The result holds for several different widely-used CF algorithms, including K-NN, an implementation of the weighted majority algorithm, and a simple popularity-based recommender.

Further experiments show that recommendation performance can sometimes be improved by augmenting user data with obviously-collected Web data. This seems to hold in several interesting situations (although not when the best-performing CF algorithm is trained on all available user data.)

The results of this paper suggest that collaborative filtering methods may be useful even in cases in which there is no explicit community of users, and in fact, no user data at all. Instead it may be possible to build useful recommendation systems that rely solely on information spidered from the Web by simple "oblivious" spiders, much like the spiders employed by search engines to index the Web. This would greatly extend the range of problems to which collaborative filtering methods could be applied.

References

[Basu et al, 1998] C. Basu, H. Hirsh, and W. Cohen. Recommendation as Classification: Combining Social and Content-Based Information in Recommendation. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*

[Cohen, 1998] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of ACM SIGMOD-98*

[Cohen, 1999] W. W. Cohen. Recognizing Structure in Web Pages using Similarity Queries. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*

[Goldman and Warmuth 1995] Sally Goldman and Manfred Warmuth. Learning binary relations using weighted majority voting. *Machine Learning*, 20:245--271, 1995.

[Good et al, 1999] N. Good, J. B. Shafer, J. A. Konstan, A. Borchers, B. Sarwar, J. Herlocker and John Riedl. Combining Collaborative Filtering with Personal Agents for Better Recommendations. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*

[Hammer et al, 1997] J. Hammer, H. Garcia-Molina, J. Cho, and A. Crespo. Extracting semistructured information from the Web. In Suciú, D., ed., *Proceedings of the Workshop on Management of Semistructured Data*, Tucson, Arizona, 1997.

[Hill et al, 1995] William Hill, Lawrence Stead, M. Rosenstein, and G. Furnas. Recommending and evaluating choices in a virtual community of use. In *Proceedings of ACM CHI'95*, pages 194--201, 1995.

[Knoblock et al, 1998] C. A. Knoblock, S. Minton, J. L. Ambite, N. Ashish, P. J. Modi, I. Muslea, A. G. Philpot, and S. Tejada. Modeling web sources for information integration. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*

[Kushmerick et al, 1997] N. Kushmerick, D. Weld, R. and Doorenbos. Wrapper induction for information extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (AAAI-98)*.

[Liebermann, 1995] H. Liebermann. Letizia: An agent that assists Web browsing. In *Proceedings of the Fourteenth International Joint Conference on Artificial International (IJCAI-95)*, Montreal, Canada.

[Mitchell, 1997] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[Muslea et al, 1998] I. Muslea, S. Minton, and C. Knoblock. Wrapper induction for semistructured, web-based information sources. In *Proceedings of the Conference on Automated Learning and Discovery (CONALD)*.

[Nakamura and Abe, 1998] A. Nakamura and N. Abe. Collaborative filtering using weighted majority prediction algorithms. In *Machine Learning: Proceedings of the Fifteenth International Conference (ICML '98)*, Madison, WI, 1998. Morgan Kaufmann.

[Perkowitz & Etzioni, 1997] M. Perkowitz and O. Etzioni. Adaptive Web sites: an AI challenge. In *Proceedings of the Fifteenth International Joint Conference on Artificial International (IJCAI-97)*.

[Resnick et al, 1994] P. Resnick, N. Iacovou, M. Sushak, P. Bergstrom, and J. Riedl. GroupLens: an open architecture for collaborative filtering of Netnews. In *Proceedings of of Computer Supported Cooperative Work Conference (CSCW)*, pages 175--186, 1994.

[Shardanand and Maes, 1995] U. Shardanand and P. Maes. Social information filtering: algorithms for automating 'word of mouth'. In *Proceedings of ACM CHI'95*, 1995.

[Soboroff et al, 1999] I. Soboroff, C. Nicholas, and M. Pazzani, editors. *Proceedings of the SIGIR-99 Workshop on Recommender Systems*, Berkeley, California, 1999.