A Dimensionality Reduction Technique for Collaborative Filtering

A. M. Roumani and D. B. Skillicorn

27 November 2006
External Technical Report
ISSN-0836-0227-
2006-527

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared December 4, 2006

Abstract

Recommender systems make suggestions about products or services based on matching known or estimated preferences of users with properties of products or services (content-based), properties of other users considered to be similar (collaborative filtering), or some hybrid approach. Collaborative filtering is widely used in E-commerce. To generate accurate recommendations in collaborative filtering, the properties of a new user must be matched with those of existing users as accurately as possible. The available data is very large, and the matching must be computed in real time. Existing heuristics are quite ineffective.

We introduce novel algorithms that use "positive" nearest-neighbor matching, that is they find neighbors whose attribute values *exceed* those of the new user. The algorithms use singular value decomposition as a dimension-reduction technique, and match in a collection of lower-dimensional spaces. Although singular value decomposition is an obvious approach to dimension reduction, it requires some care to work effectively in this setting. Performance and quality of recommendations are measured using a movie database. We show that reasonable matches can be found in time $\mathcal{O}(m \log n)$, using $\mathcal{O}(nm)$ storage space, where n is the number of users and m the number of attributes or products for which users may express preferences. This is in contrast to "approximate nearest neighbor" techniques that require either time or storage exponential in m.

1 Introduction

Recommender systems employ discovery techniques to suggest/recommend products and services during a real-time customer interaction. They are based on information filtering, interacting with users, and learning their preferences. Recommender systems have achieved widespread success in E-commerce. They are implemented today by companies that provide services such as clustering of web pages for search engine returns, proactive loading of web pages (anticipation), expertise networks, psychographics, suggestions and recommendations (e.g. movies at MovieLens.org [28], products at Amazon.com [25], eating places, newspaper articles, dating services, stocks, medical advice, and jokes).

A recommendation is based on ratings previously given by a user to products and services, and possibly on other information from the user's profile; for example, age, gender, and occupation. A recommender system's main challenge is to predict ratings for products that have not yet been seen by a user. Recommender systems employ different techniques that are usually classified, based on the recommendation approach used, into three main categories:

- Content-based approach: the user is recommended products that have commonalities with products she has rated highly in the past (e.g. in movies: actor, director, genre, language, and production company);
- Collaborative filtering (or personalization) approach: the user is recommended products that other users with similar preferences have previously liked;
- Hybrid approach: a combination of content-based and collaborative filtering methods.

We focus here on collaborative filtering, although the same algorithms can be adapted for use with content-based recommenders. The aim of collaborative filtering is to automate the process of people recommending products to one another by shifting from an individual method of recommendation to a collective one. Hence, the system must find an efficient solution to the problem of matching a user's preferences (query-tuple) against a large number of user-product ratings to determine the best recommendation. This problem is commonly known as the matching problem.

This paper presents two matching algorithms for recommender systems that use Singular Value Decomposition (SVD) [15]. The known preference data is transformed, in a preprocessing stage, so that the high-dimensional space of preferences for many possible products and services is projected into a low-dimensional space. When a recommendation is to be made, the (new) user's known preferences are mapped into the same low-dimensional space, producing values that can be rapidly compared with those of other users. The first algorithm implements this idea using a single singular value decomposition to create the low-dimensional space, while the second uses projections from randomly weighted versions of the global preference data.

An experimental system is used to evaluate the performance and quality of our algorithms, compared with other matching techniques. We show that, in practice, reasonable matches to a query can be found in time $\mathcal{O}(m \log n)$ where n is the number of users and m the number of attributes (products or services). This is in contrast to "approximate" nearest-neighbor techniques, which require either time or storage exponential in m.

Section 2 reviews existing approaches to recommender systems and discuss their capabilities and limitations. Section 3 reviews approaches to the nearest-neighbor matching

problem. Section 4 describes our new algorithms. Section 5 describes the experimental configuration. Section 6 describes the results and evaluates the performance of our algorithms compared to some baseline algorithms. Finally we draw some conclusions.

2 Existing Approaches and their Limitations

There have been many approaches to recommender systems developed in the literature since the mid 1990's, using one or more recommendation techniques.

Content-based systems, as mentioned earlier, recommend products similar to those that a user liked in the past. Hence, various candidate products must be compared with products previously rated by the user to find the best-matching product(s) and recommend them. Content-based systems work best for products with easily captured features, such as keywords in texts, or musical features in music recommending systems such as Pandora.

Many techniques for content-based recommendations have been used including information retrieval approaches [30], Bayesian classifiers [29, 13], and machine learning techniques [34]. There are also some techniques used that have been borrowed from the *text retrieval* community, such as 'adaptive filtering' [45, 49] and 'threshold setting' [40, 48].

Content-based recommender systems suffer from many limitations [2]:

- New user problem: for the system to understand and accurately match a user's preferences, the user has to rate a sufficient number of products;
- Limited content analysis: due to the limited features that are explicitly associated with products in the recommendation system;
- Over-specialization: the system cannot recommend products that are different from anything the user has rated before, since the system can only find products that score highly against a user preferences.

The collaborative filtering approach was proposed [7, 21, 47, 1] to address these limitations by computing personalized recommendations, based on other users with similar preferences. It is the best known technique today and has achieved success in computing personalized recommendations for E-commerce and information access applications, such as GroupLens [39, 23], Video Recommender [18], Ringo [44], PHOAKS system [46], Amazon.com book recommender, IBM Clever search engine [10], Jester jokes recommender [14], and QuestionBuddy recommender [11].

A wide variety of different implementations of collaborative filtering recommendation and prediction algorithms are used, for example:

- Naive Bayesian model and Bayesian Networks;
- Graph partitioning;
- Network analysis [32];
- machine learning techniques (e.g. clustering, decision trees, and artificial neural networks);
- The Average Link clustering algorithm [16];
- Clustering algorithm for categorical attributes (ROCK [17]).

The different implementations of collaborative filtering recommendation techniques can be grouped into the two general classes, heuristic-based and model-based, and in some cases a combination of the two [47, 36]. In heuristic-based algorithms [39, 44], the value of an

unknown rating for a product is computed based on the collection of products previously rated by other users, using some form of aggregate of the ratings. The aggregate, in its simplest form, can be the average of all ratings for a specific product. However, the most common approach is to compute a weighted sum of co-rated products between a user and all of his neighbours, where the weight function is essentially a distance measure representing the similarity between users. The weights help influence the prediction in favor of similar users, which allows for differentiation between user similarity levels, and hence the ability to find a set of "nearest neighbours" for each user, in addition to simplifying the rating estimation procedure. Many approaches are used to compute the similarity measure between users, where the most popular are *correlation-based*, *cosine-based*, and *mean squared difference* measure (described in [44]). Since the similarity measure is based on the intersection of co-rated products, it will not work well in computing the similarity between users whenever there are relatively few user ratings.

On the other hand, model-based algorithms [21, 1, 20, 19] predict unknown ratings based on a model, learned from the underlying dataset using statistical and machine learning techniques. For example, clustering can be applied to the user-rating matrix, and predictions computed within clusters. However, many partitioning algorithms have a bias towards equal-sized partitions or clusters with particular geometry, while hierarchical clusterers such as Average Link and ROCK [17] produce a lot of single-item clusters. Another limitation of clustering approaches is that each user can be clustered into a single category. This will affect the recommender application when users are expected to be clustered into several categories at once. Other limitations are scalability and mixed results on prediction accuracy.

A different approach is to reduce the inherently high dimensionality of the user-rating matrix and match in a lower-dimensional setting. One way to do this is proposed by Billsus and Pazzani [6] in a machine learning framework. They first transform the ratings matrix entries to boolean values representing two classes, 'like' and 'dislike', by treating each non-zero entry as '1', which does not reflect how much a user liked or disliked a product. Then a feature extraction technique (singular value decomposition) is used to reduce the boolean-matrix dimensionality, after removing all attributes (products) with less than 2 entries. The reduced space is then used to train a neural network (feed-forward neural network) to generate predictions. However, they do not provide supportive theoretical evidence for their results, nor a computational complexity evaluation.

Another dimensionality reduction approach, proposed by Sarwar *et al.* [42], suggests that sparsity should be removed in the data by filling the null entries in the ratings matrix with the average ratings for a product (or the average ratings for a user).They then use singular value decomposition to produce a low-dimensional representation of the original space. A rating for a user is predicted by regenerating that user's properties from the reduced space, but with altered ratings due to the decomposition. However, they don't provide any theoretical foundation for why the average ratings for a product (over all users) would be a good representation of the missing product rating and, if it is, then why not simply present this value as the prediction. Their other approach for generating recommendations reduces the original matrix to a low-dimensional space, then computes a neighbourhood in that space. A recommendation is made using the neighbours' opinions about products they purchased. However, the approach only considers user preference data as binary values by treating each non-zero entry as '1', which again does not reflect how much (or if) a user liked a product, but only if she consumed it or not. Furthermore, they only evaluated their work, empirically,

based on the quality of recommendations, with no regards to performance. This work is similar to one of our proposed techniques (the basic model). However our approach has no restrictions on the user-preference data types, and enables to compute recommendations in roughly linear time.

Hofmann used a statistical technique called probabilistic Latent Semantic Analysis (pLSA) [19, 20] for collaborative filtering. The approach used has some similarities with clustering methods such as distributional clustering, but differs in that the users are not partitioned into groups and user communities can overlap. It also bears close relation to dimension-reduction methods and matrix decomposition techniques, but differs in that it offers a probabilistic semantics and can build on statistical techniques for inference and model selection. However, the aspect model used in pLSA was reported to suffer from severe over-fitting problems, where the number of parameters grows linearly with the number of documents (objects). Other approaches suggested extending the traditional collaborative filtering heuristics by incorporating contextual information. For example, [1] used (in a movie recommendation application) information like when, where, and with whom a movie is seen.

For a collaborative filtering system to be accurate, it needs a large number of users to express their opinions about a relatively large number of products. Then new users' preferences need to be matched with those of existing users as accurately as possible. The available data is very large, and the matching must often be computed in real time. This creates the main challenge for existing collaborative filtering systems. Collaborative recommender systems have overcome some of the shortcomings of content-based systems. However, they still have their own limitations [5, 24], as described below.

New user problem. The same problem as for content-based systems. This problem can be addressed using hybrid recommendation approaches, or other alternative approaches [38, 47] that use techniques based on product popularity, product entropy, user personalization, or their combinations to determine the most informative products (to the system) the new user should rate;

New product problem. For a collaborative system to recommend a product, it has to be liked (rated) by a sufficient number of users. The hybrid recommendation approach can also be use to address this problem;

Sparsity. Datasets in recommender applications are usually sparse as the number of existing ratings is much smaller than the number of possible ratings. One solution to this problem is to use user profiling information when calculating similarity (sometimes called "demographic filtering" [35]), for example, gender, age, area code, and employment information. Another approach [21] to overcome the sparsity problem use associative retrieval framework and related spreading activation algorithms on users' past transactions and feedback to find transitive associations among them. A different approach was used in [42, 6] that employs a dimensionality reduction technique to reduce the dimensionality of sparse user-rating matrices;

Scalability. Currently, most recommender sites collect implicit preferences in people's actions [33]. Take for example the Amazon.com bookshop. For each book a user looks at or buys, it offers a list of related books that were bought by the same people. Peo-

ple who ordered these books have implicitly expressed their preference for the books they bought versus the books they didn't buy. However, for the collaborative filtering systems to be accurate, it needs thousands of people to express their opinions about a relatively large number of preference options (usually dozens). Computations required by existing collaborative-filtering systems in matching users' properties will dramatically increase with the (steady) growth of users and products in web-based recommender systems, resulting in serious scalability problems.

Finally, there are the hybrid approaches to recommender systems that combine content-based and collaborative methods [37, 43, 5, 27, 35], which helps to overcome some of the previously mentioned challenges. However, such approaches increase the complexity of the matching problem as they usually use more expressive rating language.

The requirements of real-time recommendations add additional challenges to the already complicated matching problem. The information filtering process in recommender systems requires an efficient matching algorithm with high throughput and scalability. For algorithms to be efficient, they have to achieve a good balance between effectiveness and performance. Clearly, the magnitude of this problem increases with respect to the number of users and products (attributes), as matches must be done in real-time. A recommender system must ensure the timely prediction of ratings upon demand.

Each of previously discussed recommendation models supports some features of recommender system, but they all have limitations, and none has the generality to efficiently find the best match between multiple matching users/products profiles.

In the next section we will discuss the matching problem in detail, and map it to the nearest-neighbor problem.

3 Nearest-Neighbor Problem

What makes matching difficult as recommender systems grow larger and more complex is that each user-tuple can consist of a potentially large number of product ratings (attributes), and a user may want recommendations for any of these attributes. Let the number of users be n and the number of attributes be m. In a recommender system setting, it is not implausible for n to be in the thousands and m to be in the hundreds. The matching problem in collaborative-filtering recommender systems is to find the user-tuple, $< r_1, r_2, ..., r_m >$ where r_i is the rating for product i, that best matches a query-tuple with the same structure, among perhaps several hundred possible matches.

There is an obvious geometric interpretation of the problem in which each user tuple and each query tuple are points in an m-dimensional space. When finding a match in recommender systems, there is little point in trying for an absolute best match, since an exact match does not provide new information to be recommended. The goal here is to find the nearest neighbor of the query tuple – but with the extra difficulty that the value of each of the user-tuple attributes must be no smaller than the value of the corresponding attribute for the query. Call this the "positive nearest-neighbor" problem. We are only interested in points that are further away from the origin than the point corresponding to the query, but we want to find, among them, the point that is nearest to the query.

3.1 Positive Nearest-Neighbor

For the positive nearest-neighbor problem, a candidate user tuple should have all of its attributes greater than or equal to those of the corresponding query attributes, and at least one attribute value is rated favorably while its corresponding attributed in the query is unrated. We refer to such a user tuple as feasible. This guarantees that a tuple contains new information to be recommended. When a feasible user-tuple is found, the system will recommend to the query user some or all of those attributes (products) that have been (highly) rated by the selected user, but not the query user.

Given a query tuple, there is an obvious brute-force algorithm for finding the best positive nearest-neighbor (user tuple) with time complexity $\mathcal{O}(nm)$. This is expensive, given that n could be very large and recommendation decisions need to be made for each query tuple submitted to the system.

3.2 Related Algorithms and Data Structures

Answering nearest-neighbor queries efficiently, especially in high dimensions, is a difficult problem. Many proposals have been made using different data structures to represent the data and clever algorithms to search them.

For a small number of dimensions m, simple solutions suffice: when $m = 1$, sorting the list of values and using binary search works effectively; when $m = 2$, computing the Voronoi diagram for the point set and then using any fast planar point location algorithm to locate the cell containing the query point also works. For larger m, say $m > 10$, the complexity of most methods grows exponentially as a function of m. Dobkin and Lipton [12] seem to be the first to give an upper bound for the time required to search for a nearest neighbor, $\mathcal{O}(2^m \log n)$ query time, and $\mathcal{O}(n^{2^{m+1}})$ preprocessing time and storage space. Most of the subsequent improvements and extensions (e.g., [8, 26, 3]) require a query time of $\Omega(f(m) \log n)$, where $f(m)$ (sometimes hidden) still denotes an exponential function of m.

One of the most widely used algorithms relies on the *k-d tree* [41], a data structure that hierarchically decomposes space into a relatively small number of cells (buckets), such that no cell contains too many objects, and providing fast access to any point by position. To search for a nearest-neighbor in a *k-d tree*, the k coordinates of the query point are used to traverse the tree until the cell containing the point is found. An exhaustive search is then used to scan the points inside the cell to identify the closest one. The average case analysis of heuristics using *k-d trees* for fixed dimension m requires $\mathcal{O}(n \log n)$ for preprocessing and $\mathcal{O}(\log n)$ query time. Although *k-d trees* are efficient in low dimensions, their query time increases exponentially with increasing dimensionality. The constant factors hidden in the asymptotic running time grow at least as fast as 2^m, depending on the distance metric used. This is because, in high dimensions, "the query hypersphere tends to intersect many adjacent buckets leading to a dramatic increase in the number of points examined" [31].

The complexity of exact nearest neighbor search led to the "approximate" nearest-neighbor problem: finding a point that may not be the nearest-neighbor to the query point, but is not significantly further away from it than the true nearest neighbor. Several approximate nearest-neighbor algorithms have been developed (e.g. [9, 22, 4]) and some were able to achieve significant improvements in running time. However, such heuristics either use substantial storage space, or have poor performance when the number of dimensions is

greater than $\log n$.

Hence, *k-d trees* and similar approaches do not seem viable, either in theory or in practice, for finding the nearest, or approximate-nearest, neighbour in high dimensions. Furthermore, searching for a 'positive' nearest-neighbour means that the performance of such algorithms becomes even worse. For example, the positive nearest-neighbour may be quite far from the query point, with many closer but infeasible objects. After the point closest to the query has been found (requiring time logarithmic in n at best), searching outwards for subsequent candidate positive-nearest neighbors can require time logarithmic in n for each step. In practice, if not asymptotically, this makes current heuristics expensive for this problem.

This suggests using a technique that can transform the high-dimensional space of the dataset into lower-dimensional subspaces. One such powerful technique is Singular Value Decomposition (SVD).

3.3 Singular Value Decomposition (SVD)

We have already noted the natural geometric interpretation of a list of tuples describing users. If we regard such a list as an $n \times m$ matrix, then the singular value decomposition can be regarded as transforming the original geometric space into a new one with the following useful property: the first axis of the new space points along the direction of maximal variation in the original data; the second axis along the direction of maximal variation remaining, and so on.

Let A be the $n \times m$ matrix representing the users. Then the singular value decomposition of matrix A is given by

$$A = USV^T \tag{1}$$

where ^{T} indicates matrix transpose. If matrix A has r linearly-independent columns (r is the rank of A), then U is an $n \times r$ orthogonal matrix (i.e. $U^T\, U = I$, identity matrix), S is an $r \times r$ positive diagonal matrix whose elements (called singular values) are non-increasing, $s_1 \geq s_2 \geq s_r > 0$, and V^T is an orthogonal $r \times m$ matrix. Each row of U gives the coordinates of the corresponding row of A in the coordinate system of the new axes (defined by V). This representation is commonly known as *thin* SVD.

The complexity of computing the SVD of a matrix is $n^2m + m^3$. Since the number of attributes, m, is typically much smaller than the number of users' tuples, n, the complexity in practice is $\mathcal{O}(n^2m)$. The space required to store the data structure is $\mathcal{O}(mr + r^2 + rn)$.

One of the most useful properties of an SVD is that the matrices on the right-hand side can be truncated by choosing the k largest singular values and the corresponding k columns of U and k rows of V^T. In particular, the truncated matrix U_k represents each user in k dimensions – but these dimensions capture as much as possible of the variation in the original data and so are a faithful representation of the high-dimensional data in fewer dimensions. Since the importance of information captured in the dimensions of U decreases as the dimensions increase, we can think of choosing $k < r$ as removing the redundant dimensions and, hence, reducing the noise in the original matrix.I In this setting, 'noise' reflects the uncertainty of users about their ratings, and the variability that they typically show from day to day.

The truncated matrices can be multiplied together as follows:

$$A_k = U_k S_k {V_k}^T \tag{2}$$

where matrix A_k is the best rank-k approximation (closest in the least squares sense) to the original matrix A. Hence the projection(s) described by an SVD are, in a strong sense, the best for spreading objects in a way that reveals their maximum variation. The problem is that typical data is both extremely sparse and close to a high-dimensional hypersphere, so that the theoretical benefits of SVD can only be turned into practical benefits by careful attention to implementation details.

This paper presents efficient matching algorithms for recommender systems. We show that a tuple of m properties can be encoded by a *single* value using SVD, given suitable normalization of the data. Matching a query tuple to appropriate user tuples requires encoding the query attributes, and then searching a ranked list of these values. We also propose a technique that produces highly accurate recommendations by using a *collection* of SVD decompositions, in which each decomposition uses data independently weighted by random scalars.This provides several different projections of the data, which tends to reveal the most important latent structure.

4 Algorithms

The difficulty of designing efficient algorithms for the nearest-neighbour problem in dimensions higher than two suggests using singular value decomposition (SVD) technique to transform high-dimensional space into a simpler low-dimensional space.

The projection(s) described by an SVD are, in a strong sense, the best for spreading objects in a way that reveals their maximum variation. Although SVD is a natural way to approach dimensionality reduction, its theoretical performance is difficult to realize in practice. The problem is that typical data is both extremely sparse and close to a high-dimensional hypersphere, so that the theoretical benefits of SVD can only be turned into practical benefits by careful attention to implementation details.

We have developed two algorithms for solving the nearest-neighbour matching problem: basic SVD-based Search (*bSVDS*), and Random-Weighted SVDS (*rwSVDS*). The algorithms integrate a ranking scheme with pruning functions, and work in two stages: a preprocessing stage and a run-time stage.

4.1 Basic SVDS (*bSVDS*)

The *bSVDS* algorithm is our basic technique. The algorithm works in two stages: a preprocessing stage and a run-time stage.

The natural similarity (proximity) metric is Euclidean distance – a user tuple is a good match for a query tuple if the Euclidean distance between them is small (and the user-tuple attributes meet or exceed the query requirements in the original space). However, it is not clear that this simple metric is the most useful in practice – it seems likely that the fit for some attributes will always be more important than for others. This could be handled by some kind of weighting of attributes (and equivalently dimensions) but the right way to do this is application-dependent and will probably require more real-world experience. However, for now, we consider all attributes equally and use Euclidean distance as our quality metric.

In algorithm (*bSVDS*) only one low-dimensional matrix is created and searched for a match to the query. We first preprocess the set of user tuples by computing the SVD of

the original matrix A, then truncate the result to one dimension (i.e. $k = 1$ in equation 2). The resulting list is sorted by increasing values of u_1. Each element of this list encodes the values of all of the attributes of the corresponding user in a manner that maximizes the variation among users – it 'spreads' them as far apart as possible along this newly created dimension.

In practice, we observed a sharp drop in the singular values after the first dimension, so we can set $k = 1$ without being concerned that we are losing major components of the preference structure.

When a query arrives at the recommender system, it must be mapped into the corresponding space of U, and a value created that can be compared to the encoded values. By re-arranging the SVD decomposition equation we get:

$$U = AVS^{-1} \tag{3}$$

In other words, points of A can be mapped into the transformed space by multiplying them by VS^{-1}. This same multiplication can be applied to query tuples to compute their coordinates in the transformed space. Since we have truncated the SVD at $k = 1$, this mapping requires only the first column of V and the first singular value, and therefore takes time $\mathcal{O}(m)$.

After the transformation maps the query tuple to a single value, the value is compared to the user values using binary search to find the user with the closest value. This user's tuple may not be feasible (it is similar to the query tuple in the original A matrix but one or more of its attributes is smaller than the corresponding requirement of the query). In this case, the ranked list is searched in a zigzag fashion from the original entry, by choosing the next closest value on either side of it, until a feasible user tuple is found.

4.2 Random-Weighted SVDS (*rwSVDS*)

Algorithm *rwSVDS* is an extension to *bSVDS*; instead of using a single search list to predict the positive nearest-neighbor point, it uses multiple search lists. The global nearest neighbor is derived from the search result over all lists.

Weighting of attributes offsets their importance, and hence changes the main variations captured in the first dimensions of the decomposition. If we can understand how the random scatter of the data arise in our experimental system, we may be able to calculate the appropriate weighting factors based on theory. However, if we expect the random scatter of the data to vary along the distribution curve, then we may weight points differentially.

The random-weighted SVDS (*rwSVDS*) uses a set of 3 decision lists to predict the nearest-neighbour point. In a dataset A, let the number of objects be n and the number of attributes of an object be m. Each search list in *rwSVDS* is constructed using the following algorithm:

- Create a vector w of size m of random weighting scalars in (0..1];
- Multiply (dot product) vector w into rows of matrix A. This minimizes the quantities in the columns differently;
- Project the resulting matrix into low dimensions using the SVD technique to generate the U space;
- Truncate U to one-dimensional space;
- Rank the list by sorting.

This process is repeated 3 times to generate 3 one-dimensional search lists, based on the differently weighted matrices.

When a query is submitted to the system, the same process is applied to it to correspond to each search list.T The same w weighting vector that was used on the objects in a certain list, is used on the query to generate the weighted tuple. Then the selection process transforms each query weighted tuple into the U space of each SVD, then searches all 3 ranked lists in a concurrent fashion to find a common match. The first feasible user tuple to have been found on all 3 lists is reported as the best match.

These algorithms could easily be extended to include user profile data such as age, gender, geography region, and zip codes.

5 Experimental Configuration

5.1 Dataset

In our experiments we used a dataset collected from MovieLens. We use the most extensive dataset available for download. Each user used in our experiments has at least 20 movie ratings (attributes) in the original dataset. Ratings are made on a 5-star scale (whole-star ratings only), with larger values denoting higher appreciation, and 0 indicating an unrated movie. Each result is the average over 70 runs.

5.2 Baseline Algorithms

For comparison purposes, we also consider a ranking algorithm that uses the sum of the attributes as the value for ranking. We call it the SUM-based Search algorithm (*SUMS*). The advantage of the sum is that any user tuple whose sum is smaller than the sum of the requirements of a query tuple cannot possibly be feasible. We compute the sum of attributes for each user tuple and sort the list based on the sum of ratings. The sum is also computed for each query tuple. Then binary search is used to find the user tuple closest in sum to it, and a feasible user-tuple is searched for in the manner described above. Algorithm *SUMS* will in practice perform best when the magnitudes of typical attributes are about the same – this can be arranged by normalizing if necessary.

Both our *SVDS* algorithms and *SUMS* algorithm have similar properties: both require $\mathcal{O}(nm)$ storage for the ranking information (since the full set of attributes must be checked for feasibility); for both, the cost of binary search is $\mathcal{O}(\log n)$, and for both the cost of computing the fit between a query tuple and a user tuple is $\mathcal{O}(m)$. The preprocessing required for *SVDS* is more expensive. However, this cost is amortized over all the matches of queries to users. The performance difference between the two rankings depends on how many list elements must be examined to find a feasible match, and on the quality of such a match. The best way to assess the computational tasks associated with finding a feasible user tuple is to observe the actual runs and analyze the quality of the results, as we will do in Section 6.

We also compare our algorithms' effectiveness with that of randomly selecting users until a feasible one is found (call this simple algorithm *RAND*), and with exhaustive search. *RAND* provides a baseline for the number of probes required to find a good solution, while exhaustive search provides a baseline for how good a solution is possible.

5.3 Evaluation Metrics

To assess the quality of a solution, we use the Euclidean distance as the metric for measuring proximity to the query point. We determine the number of probes required to search for a matching user-point, varying the following parameters:

- Number of users
- Number of ratings

The main performance measures of interest are the cost of finding a match, and the quality of this match:

- Cost – measured in number of probes needed to find a match, including the cost of binary search (where applicable);
- Sub-Quality – the Euclidean distance from the match point found by the algorithm to the query point;
- Sub-Optimality-Ratio – the ratio of the solution found by an algorithm to the optimal solution;

For each combination of experiments, the fraction of user tuples that are feasible (satisfying positive nearest-neighbor) for a query are held to approximately 5%. This, in our view, models the most plausible scenario. If feasible matches are extremely scarce, then exhaustive search is the best matching technique, although users may find such a system too frustrating to use because of the delay. On the other hand, if the fraction of feasible users is large, then the system is hugely under-utilized which is also an unlikely scenario.

The feasibility fraction is forced by generating query tuples in the following way. Approximately 5% of the user tuples are selected at random and their pointwise minimum is used as a query tuple. This query tuple, by construction, is feasible for at least 5% of users but could, of course, be feasible for a much larger fraction. We aim to create and use queries that are feasible for between 3 and 7% of user tuples, so if the feasibility is too high, a subset of the current user tuples are selected, and the process repeated until the feasibility percentage falls into this range.

5.4 Normalization

Our experience has been that SVD, despite its theoretical determination of the optimal projection of the data, does not perform well in practice without careful normalization. Normalization is a process of centering and scaling the dataset. We normalize the data by zero-centering each column using fixed mean 3, as this is the middle point of the rating range. The fixed mean is subtracted from the non-zero entries only. By this normalization, neutral opinions are treated the same as a non-opinion (expressed originally as 0). This behavior is plausible since it means that both possibilities are interpreted as providing no extra information about a movie's quality. The net effect is to add more weight to strongly positive and negative opinions.

When summing the attributes in the *SUMS* algorithm, neutral values are treated as zeros. Also when assessing the quality of a solution, the Euclidean distance to the query point is calculated based on the zero-centered values instead of the original raw values.

6 Results and Discussion

We now study the effect of varying the number of users and attributes (movies) on the search cost and quality of the solutions given by different algorithms. First, we compare the random-weighted SVDS algorithm (*rwSVDS*) to the *RAND* and *SUMS* algorithms.

Figure 1 plots the number of probes (left column) required to find a positive-nearest-neighbor (feasible) user-tuple for each query, and the sub-quality of this match (right column). Plots (a) and (b) show the number of probes required by *rwSVDS* and the sub-quality for a range of users and attribute, plots (c) and (d) show the same for *SUMS*, and plots (e) and (f) show those for *RAND*.

It is clear from the figures that algorithm *rwSVDS* achieves much better results than *RAND* and *SUMS*. It requires, on average, fewer than half of the probes that *SUMS* requires, with the peak performance at around 100 users and 40 attributes, where it requires only a fifth as many probes. It also outperforms *RAND*, requiring about a fourth of the probes on average, and less than a tenth of the probes for small numbers of users and attributes.

Algorithm *SUMS*, on the other hand, performs better than *RAND* by an average of 20% fewer probes in almost all settings, peaking at 50% fewer probes for small number of users. However, for large numbers of users and attributes, *SUMS* falls behind *RAND* by requiring about 15% more probes.

We observe that, as the number of users and attributes increase, the search cost increases for all algorithms, but very slowly for *rwSVDS*, as it becomes harder to find a solution point with the required attributes.

As for the quality of the solution, Figures 1(b), 1(d), and 1(f) show that *rwSVDS* finds better-quality matches than *RAND* and *SUMS* for all parameter settings – an average of 4.6 times better quality than *SUMS*, and 16 times better than *RAND*. Although *SUMS* requires many more probes and finds lower-quality matches than *rwSVDS*, it still finds better-quality matches than *RAND* for all parameter settings.

Figure 2 plots the sub-optimality ratio of the solution found by *rwSVDS* in comparison to the optimal solution point. This ratio is close to 1, showing that *rwSVDS* achieves almost optimal solutions. Although the sub-quality for *rwSVDS* (Figure 1(b)) increases slightly with the number of attributes, the sub-optimality ratio stays the same. This is due to the increased Euclidean distance to the query point, even for the optimal match.

For algorithm *bSVDS*, Figure 3 plots the number of probes (left column) and the sub-quality (right column). Algorithm *bSVDS* requires twice as many probes as *rwSVDS* in most settings, except for small numbers of attributes. However, it maintains a superior performance compared to *SUMS* and *RAND*. As for the match quality, there is no significant difference from *rwSVDS*.

From the results above, we see that the random-weighted SVDS (*rwSVDS*) achieves the best-quality matches of near optimal and better overall performance compared to *bSVDS*. In comparison to *SUMS* and *RAND* it also achieved, by far, better cost and better solution quality.

7 Conclusions

There are several critical research challenges in the development of recommender systems that support services for web-based interactive applications. One of these challenges is the

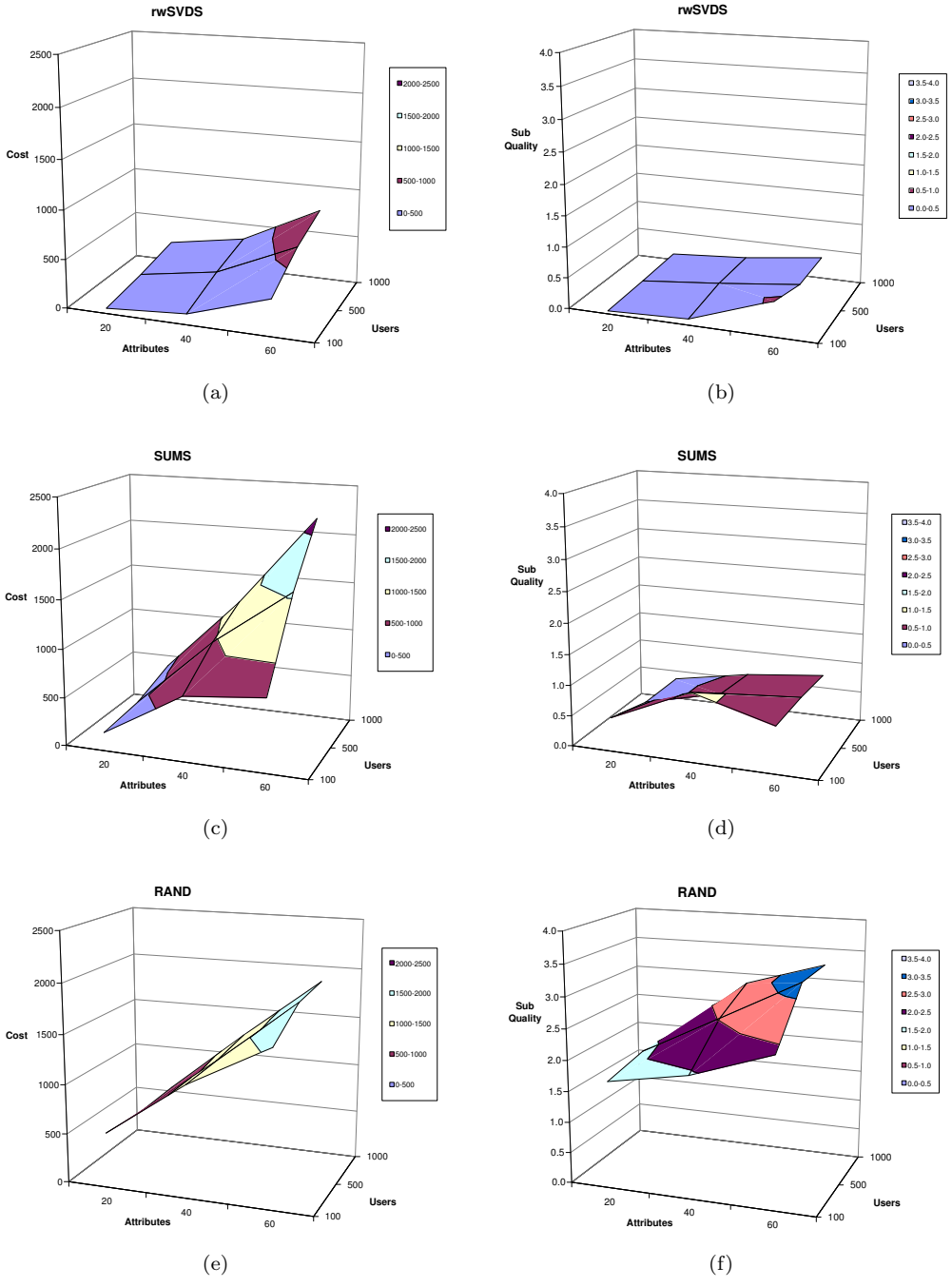
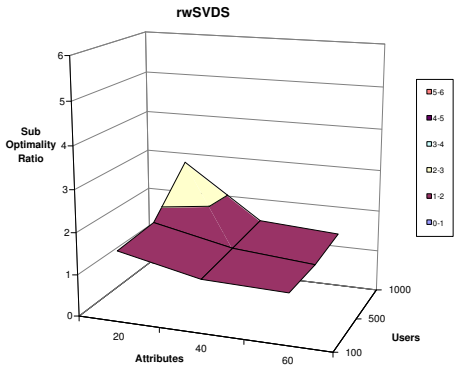


Figure 1: Search cost and sub-quality for: (a) and (b) *rwSVDS*, (c) and (d) *SUMS*, (e) and (f) *RAND*



(a)

Figure 2: Sub-optimality Ratio for *rwSVDS*

discovery of an effective matching algorithm that is scalable and reasonably effective in providing good matches between user-query preference criteria and available user/product profiles. As the number of users and products increase, solutions that are cheap to implement but still provide reasonably good matches are required. To our knowledge, none of the existing matching algorithms is an attractive candidate.

The apparent difficulty in designing effective (approximate) nearest-neighbour techniques that are efficient in the worst-case runtime, with respect to both query time and space, comes from having high dimensions. We proposed using a technique that can transform a high-dimensional space into a lower dimensional space, namely Singular Value Decomposition (SVD).

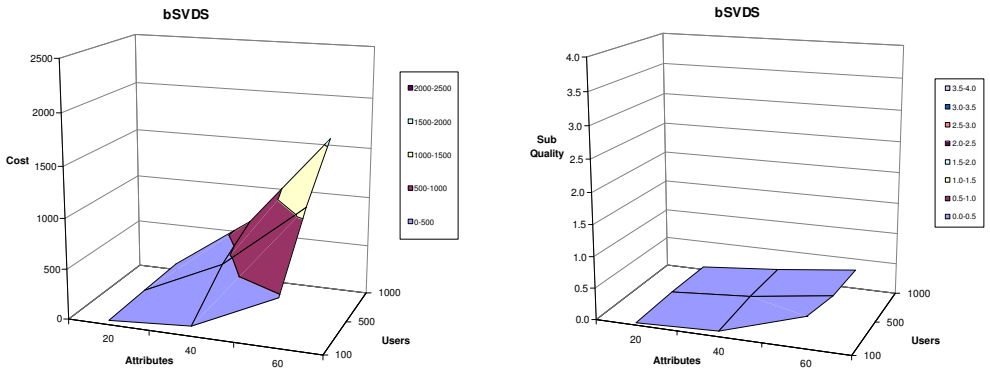
Singular value decomposition (SVD) is a natural way to approach dimensionality reduction, but its theoretical performance is difficult to realize in practice. This approach does not work well when applied naively.

We have presented two collaborative filtering techniques based on SVD. Our algorithms use SVD as a preprocessing step to project user properties into low-dimensional spaces. Careful normalization, and the use of multiple projections based on random weighting of attributes result in one-dimensional lists that can be searched, in practice, in only a constant number of probes beyond the basic binary search required to find the right part of the list. The overall complexity of matching is $\mathcal{O}(m \log n)$, even for the positive nearest neighbor case.

References

[1] G. Adomavicius, R. Sankaranarayanan, S. Sen, and A. Tuzhilin. Incorporating contextual information in recommender systems using a multidimensional approach. *ACM Trans. Inf. Syst.*, 23(1):103–145, 2005.

[2] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on*



(a) Cost (b) Sub-Quality

Figure 3: Search cost and sub-quality for *bSVDS*

Knowledge and Data Engineering, 17(6):734–749, Jun. 2005.

[3] P. K. Agarwal and J. Matousek. Ray shooting and parametric search. In *STOC'92: Proc. 24th ACM Symp. On Theory of Computing*, pages 517–526, 1992.

[4] S. Arya, D. M. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[5] M. Balabanovi and Y. Shoham. Fab: Content-based, collaborative recommendation. *Communications of the ACM*, 40(3):66–72, 1997.

[6] D. Billsus and M. J. Pazzani. Learning collaborative information filters. In *Proc. 15th Int. Conf. on Machine Learning*, pages 46–54. Morgan Kaufmann, San Francisco, CA, 1998.

[7] J. Canny. Collaborative filtering with privacy. In *IEEE Symposium on Security and Privacy*, pages 45–57, Oakland, CA, May 2002.

[8] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Computing*, 17(4):830–847, 1988.

[9] K. L. Clarkson. An algorithm for approximate closest-point queries. In *SCG'94: Proc. 10th Annual symposium on Computational geometry*, pages 160–164, 1994.

[10] CLEVER search engine, Sept. 2006. `alme1.almaden.ibm.com/cs/k53/clever.html`.

[11] W. M. Davies and H. C. Davis. QuestionBuddy – a collaborative question search and play portal. In *Proc. 10th Int. CAA Conference (in press)*, 2006.

[12] D. Dobkin and R. Lipton. Multidimensional search problems. *SIAM Journal on Computing*, 5:181–186, 1976.

[13] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[14] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151, 2001.

[15] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA, second edition, 1989.

[16] E. Gose, R. Johnsonbaugh, and S. Jost. *Pattern Recognition and Image Analysis*. Prentice Hall, 1996.

[17] S. Guha, R. Rastogi, and K. Shim. ROCK: A robust clustering algorithm for categorical attributes. In *Proc. 15th Int. Conf. On Data Eng.*, 1999.

[18] W. Hill, L. Stead, M. Rosenstein, and G. Furnas. Recommending and evaluating choices in a virtual community of use. In *Proc. SIGCHI Conf. on Human Factors in Computing Systems*, pages 194–201, 1995.

[19] T. Hofmann. Unsupervised learning by probabilistic latent semantic analysis. *Machine Learning*, 42(1-2):177–196, 2001.

[20] T. Hofmann. Latent semantic models for collaborative filtering. *ACM Trans. Inf. Syst.*, 22(1):89–115, 2004.

[21] Z. Huang, H. Chen, and D. Zeng. Applying associative retrieval techniques to alleviate the sparsity problem in collaborative filtering. *ACM Trans. Inf. Syst.*, 22(1):116–142, 2004.

[22] J. Kleinberg. Two algorithms for nearest-neighbour search in high dimensions. In *Proc. 29th Annual ACM Symposium. on Theory of Computing*, pages 599–608, May 1997.

[23] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl. GroupLens: Applying collaborative filtering to usenet news. *Commun. ACM*, 40(3):77–87, 1997.

[24] W. S. Lee. Collaborative learning for recommender systems. In *Proc. 18th Int. Conf. on Machine Learning*, pages 314–321. Morgan Kaufmann, San Francisco, CA, 2001.

[25] G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[26] J. Matousek. Reporting points in half-spaces. In *Proc. 35th Symposium on Foundations of Computer Science*, pages 207–215, 1991.

[27] P. Melville, R. Mooney, and R. Nagarajan. Content-boosted collaborative filtering for improved recommendations. In *AAAI'02: Proc. 18th National Conf. on Artificial Intelligence*, Jul. 2002.

[28] B. N. Miller, I. Albert, S. K. Lam, J. A. Konstan, and J. Riedl. MovieLens unplugged: Experiences with an occasionally connected recommender system. In *IUI'03: Proc. 8th Int. Conf. on Intelligent User Interfaces*, pages 263–266, Miami, Florida, USA, 2003.

[29] R. Mooney, P. Bennett, and L. Roy. Book recommending using text categorization with extracted information. In *Proc. AAAI Workshop on Recommender Systems*, pages 70–74, Madison, WI: AAAI Press, 1998.

[30] R. J. Mooney and L. Roy. Content-based book recommending using learning for text categorization. In *Proc. of DL-00, 5th ACM Conf. on Digital Libraries*, pages 195–204, San Antonio, US, 2000.

[31] S. A. Nene and S. K. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19(9):989–1003, 1997.

[32] Network analysis, Sept. 2006. `www.orgnet.com/buzznet.html`.

[33] D. M. Nichols. Implicit rating and filtering. In *Proc. 5th DELOS Workshop on Filtering and Collaborative Filtering*, pages 31–36, Budapest, Hungary, Nov. 1997.

[34] M. Pazzani and D. Billsus. Learning and revising user profiles: The identification of interesting web sites. *Machine Learning*, 27(3):313–331, 1997.

[35] M. J. Pazzani. A framework for collaborative, content-based and demographic filtering. *Artificial Intelligence Review*, 13(5-6):393–408, 1999.

[36] D. Pennock, E. Horvitz, S. Lawrence, and C. L. Giles. Collaborative filtering by personality diagnosis: A hybrid memory- and model-based approach. In *UAI'00: Proc. 16th Conf. on Uncertainty in Artificial Intelligence*, pages 473–480, Stanford, CA, 2000.

[37] A. Popescul, L. H. Ungar, D. M. Pennock, and S. Lawrence. Probabilistic models for unified collaborative and content-based recommendation in sparse-data environments. In *Proc. 17th Conf. in Uncertainty in Artificial Intelligence*, pages 437–444, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[38] A. M. Rashid, I. Albert, D. Cosley, S. K. Lam, S. M. McNee, J. A. Konstan, and J. Riedl. Getting to know you: Learning new user preferences in recommender systems. In *Proc. 7th Int. Conf. on Intelligent User Interfaces*, pages 127–134, New York, NY, USA, 2002. ACM Press.

[39] P. Resnick, N. Iacovou, M. Suchak, P. Bergstorm, and J. Riedl. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, pages 175–186, Chapel Hill, North Carolina, 1994.

[40] S. Robertson and S. Walker. Threshold setting in adaptive filtering. *Journal of Documentation*, 56(3):312–331, 2000.

[41] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[42] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Application of dimensionality reduction in recommender systems–a case study. In *ACM WebKDD Workshop*, 2000.

[43] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock. Methods and metrics for cold-start recommendations. In *Proc. 25th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 253–260, 2002.

[44] U. Shardanand and P. Maes. Social information filtering: Algorithms for automating "word of mouth". In *CHI'95: Proc. of ACM Conf. on Human Factors in Computing Systems*, volume 1, pages 210–217, 1995.

[45] G. Somlo and A. E. Howe. Adaptive lightweight text filtering. In *Proc. 4th Int. Conf. on Advances in Intelligent Data Analysis*, pages 319–329, London, UK, 2001. Springer-Verlag.

[46] L. Terveen, W. Hill, B. Amento, D. McDonald, and J. Creter. PHOAKS: A system for sharing recommendations, Mar. 1997.

[47] K. Yu, A. Schwaighofer, V. Tresp, X. Xu, and H.-P. Kriegel. Probabilistic memory-based collaborative filtering. *IEEE Transactions on Knowledge and Data Engineering*, 16(1):56–69, 2004.

[48] Y. Zhang and J. Callan. Maximum likelihood estimation for filtering thresholds. In *Proc. 24th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 294–302, 2001.

[49] Y. Zhang, J. Callan, and T. Minka. Novelty and redundancy detection in adaptive filtering. In *Proc. 25th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 81–88, 2002.