

# **Contributions to Collaborative Filtering Research**

**Shuzhen Nong**

**Major Professor: Jon Herlocker**

**Committee Members: Margaret Burnett**

**Francisco Martin**

## ABSTRACT

In this report, I first introduced ~~the~~ three areas ~~that of interest to~~ collaborative filtering researchers ~~are interested in and challenged by~~, namely: (a) how to solve the sparsity and scalability problems in recommendation systems; (b) how to ~~synthesize various~~ rapidly develop and test collaborative filtering algorithms; and (c) how to apply collaborative filtering to ~~implicit preference data~~ unbounded numeric preference data. In the next three chapters, I described my contributions to these three areas: (a) my work on the CoFE recommendation engine; (b) my work on the SVD based algorithm; and (c) my attempt to design a new algorithm for the iTunes play count data. A summary and prospects for future work are given in the final chapter.

### **Acknowledgements**

I would like to express my sincere gratitude to my advisor, Dr. Jon Herlocker, for his support during my two years study in computer science. It was he who gave me the opportunity to get to know the collaborative filtering and data mining. It was this opportunity that made me realize what I am interested in and would like to dedicate myself to. I am also grateful for his guidance during the writing phases of this report.

I would also like to thank Dr. Francisco Martin for his support during my work on the iTunes recommendations project. His advice helped me not only to think deep into the algorithms, but also to grasp the big picture from the business perspectives.

I would also like to thank Dr. Margaret Burnett for serving in my committee. I am also grateful for her support and encouragement.

Thanks also go to my group members. I could not have had these project results without their teamwork.

Finally, I would like to thank my husband for his understanding and support, but my appreciation would go beyond words.

## Table of Contents

1	Introduction.....	6
1.1	Overview of collaborative filtering.....	6
1.2	Three areas in collaborative filtering that need further research.....	7
1.3	My contributions.....	10
2	Support for CoFE recommendation system in multiprocessing, performance tests and autotest mode.....	12
2.1	Introduction to CoFE.....	12
2.2	My work on the server mode.....	13
2.3	My work on the autotest mode.....	16
3	Implementation and evaluation of an improved recommendation algorithm based on Singular Value Decomposition.....	18
3.1	Advantages of dimensionality-reducing algorithms.....	19
3.2	Linear algebra of SVD.....	20
3.3	Application of SVD technique to movie recommendations.....	21
3.3.1	SVD decomposition.....	21
3.3.2	Application 1: Directly use the truncated rating matrix as a prediction matrix.....	23
3.3.3	Application 2: Use SVD for neighborhood selections.....	23
3.4	Evaluation of SVD based algorithms.....	26
3.5	Conclusions and future work.....	29
4	Design and implementation of a new recommendation algorithm that applies to iTunes play count data.....	30
4.1	Analysis of the play count data.....	31
4.2	Challenges in recommendation algorithms and an ideal algorithm.....	35
4.2.1	Challenges in recommendation algorithms.....	35
4.2.2	An ideal algorithm.....	39
4.3	The initial proposed top N recommendation algorithm.....	39
4.3.1	Karypis' [17] algorithm.....	40
4.3.2	My modification.....	41
4.3.3	A proposal on pre-clustering.....	44
4.3.4	Summary of the proposed recommendation algorithm.....	45
4.3.5	Criticism of this algorithm.....	45
4.4	Summary and future work.....	46
5	Summary and future work.....	48
6	References.....	49
7	Appendix.....	51

### List of Figures

<b>Figure 1:</b> The first 14 eigenvalues of the MovieLens data set.....	22
<b>Figure 2:</b> MAE comparison between Simple Pearson and SVD-based algorithms.....	27
<b>Figure 3:</b> The number-frequency law for User1.....	32
<b>Figure 4:</b> The log-log plot of the number-frequency law for User1.....	32
<b>Figure 5:</b> The rank-frequency law for User1.....	33
<b>Figure 6:</b> The log-log plot of the rank-frequency law for User1.....	33
<b>Figure 7:</b> The rank-frequency law for all users.....	34
<b>Figure 8:</b> The log-log plot of the rank-frequency law for all users.....	34

### List of Tables

<b>Table 1:</b> MAEs of Simple Pearson, Simple SVD, and SVD+neighborhood algorithms. The best score in each row is in bold. When the size of the test set is small, SVD+Cosine performs best. With larger amounts of test data, Simple Pearson performs best.....	26
<b>Table 2:</b> Summaries of the eight iTunes users' descriptive statistics (of their playcounts).....	31
<b>Table 3:</b> The similarities of item2 and item8 with all items and their play counts.....	42
<b>Table 4:</b> The similarities of item2 and item8 with all items and their play counts and weights.....	43
<b>Table 5:</b> Recommendation scores from two algorithms.....	43

## Contributions to Collaborative Filtering Research

### 1. Introduction

#### 1.

In this chapter, I will first introduce the concept of collaborative filtering. Next, I list ~~the~~ three areas in collaborative filtering ~~that we need to do~~ requiring more research ~~on~~. Then I list my contributions in these three areas. Finally, I introduce the organization of the rest of this report.

### 1.1 Overview of collaborative filtering

As the information on the internet increases exponentially, we have experienced the phenomenon of “information overload”—the volume of books, movies, and songs is growing explosively. The amount of information is more than any person can possibly filter through in order to find what one wants or needs. Collaborative filtering techniques aim at solving these information overload problems.

Collaborative filtering systems predict a user’s interest in new items based on the recommendations of other people with similar interests. Instead of performing content indexing or content analysis, collaborative filtering systems rely entirely on interest ratings from members of a participating community.

As Herlocker [12] pointed out, compared with content-based filtering techniques, collaborative filtering has the following advantages: (a) “Users can filter information of any type of content... Content-based filtering has the limitations that limits it to well-structured content that can be compactly summarized in electronic format, while collaborative filtering can be used to filter more complex parameters, such as art work, music, and vacation packages”; (b) “Users can filter based on complex and hard to represent preference, taste and quality”; (c) “Users can receive serendipitous recommendations, which are not necessarily similar to what they have already consumed.”

Formatted: Indent: Left: 0",  
Numbered + Level: 1 + Numbering  
Style: 1, 2, 3, ... + Start at: 1  
+ Alignment: Left + Aligned at:  
0.25" + Tab after: 0.5" +  
Indent at: 0.5", Tab stops:

Many algorithms have been proposed and tested on real-world recommendation problems and many recommendation systems have been developed. One of the best-known recommendation systems ~~is would be that of~~ Amazon.com, which uses recommendation algorithms to personalize an online store for each customer. Other popular recommendation ~~algorithms systems~~ include Ringo, a personalized music recommendation system using user's ratings for music, and MovieLens, a recommendation system that ~~aims to~~ recommends movies to users.

### 1.2 Three areas in collaborative filtering that need further research

Three ~~areas issues are currently that currently~~ challenging collaborative filtering researchers are:

(a) **Sparsity and scalability problem**

Recommendation systems have to deal with huge amounts of data, tens of millions of customers, and millions of distinct catalog items. As Linden et al [18] noted when summarizing their algorithms for Amazon's recommendation engine, "*Recommendation applications require the results set to be returned in real time, in no more than half a second, while still producing high-quality recommendations.*" This statement suggests that online speed and accuracy are two important requirements for recommendation systems.

The sparsity problem is challenging recommendation systems' speed and accuracy. Since there might be millions of distinct items, usually a user can only rate a small percentage of all items. Thus, sparsity is an integrated problem of collaborative filtering. For example, in EachMovie data set, which can be represented as a matrix with 943 rows (users) and 1873 columns (movies), there are only 100,000 ratings—only 5.66% of cells in the matrix have ratings. In the MovieLens data set, there are 6,040 users, 3,900 movies, 1,000,209 ratings—only 4.25% of cells in the user-movie matrix have ratings. Sparsity might also cost some algorithms their coverage, which measures the percentage

Formatted: Indent: Left: 0"

Formatted: Indent: First line: 0"

Formatted: Font: Bold

Formatted: Indent: Left: 0"

of ratings an algorithm can predict. For example, we cannot compute two users' correlation if there are no co-rated items between them.

Another challenge is to improve the scalability of the recommendation algorithms. For example, some current algorithms recommend items to an active user through finding neighbors (the users that are similar to the active user). They are able to search tens of thousands of potential neighbors in real-time, but modern systems demand the ability to search tens of millions of potential neighbors [18]. We need recommendation systems that are able to maintain speed when the number of users increases.

One possible way to solve the scalability problem is sampling. We can reduce the data size by selecting only those users with a large amount of information (by using statistics such as entropy) and use only these data to run our recommendation system [9].

However, if we only examine a small sample, ~~as with~~ the number of users become lower, less potential neighbors ~~there~~ will be in the sample, ~~so~~ we might lose the opportunity to select the best neighborhood for that user [18].

There are two other possible approaches to the scalability problem. One is to explore data condensing and dimension reducing methods, which could reduce the size of data, while still keeping the largest amount of information. The other is to explore the possibility of moving some online computation offline. The users' experience is generally not impacted by increases in offline computation. So, if we could move some computation offline and only update that offline computation periodically, ~~then~~ the online computation would become less and could result in faster online services. The key to this method is to guarantee that the delay of that offline computation does not affect the accuracy of the recommendation. In the first approach, we might also compute the data condensing offline, and in this respect, these two approaches are similar. However, the key to the second approach is to move computation that is relatively stable offline, while maintaining the accuracy of the recommendation [18].

(b) **How to rapidly develop and test collaborative filtering algorithms** Synthesis of algorithms—the challenge of finding the “best” recommendation algorithm

Formatted: Font: Bold

Formatted: Indent: Left: 0"

Formatted: Normal

Researchers have been searching for the most accurate and fastest algorithm with which to solve collaborative filtering problems. While some authors claim that they have solved the sparsity and scalability problems and have achieved higher accuracy and greater throughputs, we find their results hard to generalize beyond the context of their work, due to differences in experimental procedures, datasets, and evaluation methods across studies.

Perhaps one way to address this problem is to summarize and “synthesize” all of the collaborative filtering algorithms. In other words, rather than continue to propose new algorithms using methodologies that inhibit cross comparison, we could seek to synthesize the diversity of work that has been done before into a coherent picture [5].

For this purpose, we need to build up a highly extensible software framework for investigating collaborative filtering algorithms and enabling rapid design and evaluation of new algorithms. This software system should be able to standardize the data sets of different algorithm experiments, their evaluation metrics, and their experimental protocols, including different methods for splitting training data and test data, different averaging methods when applying evaluation metrics, and treatment of recommendations for which test ratings are not available.

(c) **How to apply collaborative filtering to unbounded and numeric preference data**

Formatted: Indent: Left: 0"

Formatted: Font: Bold

How to apply collaborative filtering to implicit preference data

Traditional collaborative filtering is applied to “explicit ratings”—the rated preferences that are explicitly expressed by users. To be reliable, the system needs a very large number of people to express their preferences about a relatively large number of options. This requires quite a lot of effort from a number of people. Since the system only

becomes useful after a certain number of opinions have been collected, people may not be very motivated to express detailed preferences in the beginning stages, when the system cannot yet provide them with the most accurate recommendations.

Another application domain for collaborative filtering has been to apply it to “implicit preference data”—data that represent users’ preferences that are collected implicitly while the users are using the system. For example, people who order books from an Internet bookshop implicitly express their preference for the books they buy over the books they do not buy. Customers who have bought the same book are likely to have similar preferences for other books as well.

There are challenges in applying collaborative filtering to implicit rating data. First, as implicit rating data are collected automatically every time users employ the system, we can collect more data. Thus, there are challenges in storing and processing the data efficiently. The scalability problem will be more challenging. Second, implicit rating might be very “noisy,” as there might be times when the system does not function properly or users manipulate the system maliciously. Previous collaborative filtering has focused on preference data that has a fixed number of possible values, such as purchase data (unary) and explicit ratings data (often 1-5, 1-7, or 1-10). On the other hand, no published work deals withThird, implicit ratings are usually frequency counting data with unlimited upper bounds, as opposed to explicit ratings with limited scales. Normalizing unbounded data would be a problem in applying collaborative filtering to implicit rating data.

### 1.3 My contributions

During my research assistantship in collaborative filtering, I have been working on the following tasks:

- (a) improvements to the CoFE recommendation system—a collaborative filtering research system that aims at providing a framework for rapid turn-around research into collaborative filtering algorithms—in client/server development,

multiprocessing supporting, performance testing, and dynamic loading of  
algorithm classes ~~dynamic loading~~;

(b) implementation and evaluation of an recommendation algorithm based on  
Singular Value Decomposition which tries to address the sparsity and scalability  
problems,

(c) design and implementation of a new recommendation algorithm that applies to  
music play count data.

The following report is organized according to my contributions. Chapter 2 is a summary  
of my contribution to the CoFE system. Chapter 3 is a description of my SVD algorithm  
experiments. Chapter 4 summarizes my understanding and first attempts to explore the  
music recommendation algorithm. A summary and prospects for future work are given in  
Chapter 5.

Formatted: Indent: Left: 0"

## 2 Support for CoFE recommendation system in multiprocessing, performance tests, and automated algorithm testing~~autotest mode~~

### 2.1 Introduction to CoFE

Collaborative filtering recommendation systems are software systems that enable communities to perform collaborative filtering. At the center of a collaborative filtering system is a collaborative filtering recommendation engine—the software system responsible for the computation involved in collaborative filtering. The recommendation engine is responsible for analyzing ratings, determining who is a neighbor to whom, and computing the predictions and recommendations. ~~The~~ CoFE is one such recommendation engine.

There are two different purposes for CoFE, one as a research tool for computer scientists (we call this the CoFE autotest mode, because it enables them to automatically test algorithms), and the other as a potential production-level backend for a collaborative filtering system (we call this the CoFE server mode).

As a research tool, CoFE goals are to allow researchers to easily create and evaluate new algorithms on standard datasets. As a collaborative filtering backend, CoFE's goals are to ~~The following are the functions of CoFE listed in the CoFE page (<http://cees.oregonstate.edu/iis/twiki/bin/view/IIS/CoFEFunctionRequirements>).~~

#### As a research tool

- ~~○Be able to let researchers plug-in and tweak new algorithms~~
- ~~○Be able to test algorithms' performances on standard datasets~~
- ~~○Be able to present those tests to a user test easily.~~
- ~~○Be able to publish those test results easily.~~

#### As a collaborative filtering system backend

- ~~○Be able to connect with other e-business systems with some RPC technologies (like RMI).~~
- ~~○Be able to deliver fast prediction and recommendation~~

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

~~○Be able to serve different user sessions with different algorithms or algorithms parameters.~~

During my work on CoFE, I have worked with both the server mode and the autotest mode.

## 2.2 My work on the server mode

I was responsible for CoFE's client/server development and maintenance, support for multiprocessing, and performance testing. In particular, I will discuss the support for multiprocessing and performance testing.

### (a) Support for multiprocessing

CoFE is written in Java. The server mode supports multi-user real time recommendations, and allows concurrently running ~~user~~ threads, each computing recommendations or predictions. As a result, we need to identify all fields in objects that are shared and make sure that we provide some form of synchronization.

There are two kinds of potential problems that might be caused by inappropriate synchronizations: data inconsistency and deadlock.

The problem of data inconsistency is caused by lack of synchronization for critical data sections. As a result, different threads may incorrectly overwrite a shared variable. For example, we store the cached ratings into a data structure called "ratingsMatrix". All threads that add ratings, remove ratings, or compute predictions for items and Top N recommendations must access this data structure. If, at the time one thread is adding ratings to the data structure, another thread is retrieving ratings for computation, data inconsistency might arise, particularly if the reader reads data that is the result of only a partial write. Thus, this data structure has to be synchronized.

The other problem caused by inappropriate synchronization is deadlock. Deadlock results from “over” synchronization. For example, there are two objects that are protected by lock A and B. thread t1 is holding lock A, while waiting for thread t2 to release the lock B. Meanwhile, thread t2 is holding lock B, while waiting for thread t1 to release the lock A. In this situation, no threads can move on, and a deadlock is created.

CoFE utilizes the synchronization mechanism provided by the Java language itself. As introduced in [24], “*Java implements an elegant twist on an age-old model of interprocess synchronization: the monitor.*” The monitor is a control mechanism like a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, the monitor can be used to protect a shared asset from being manipulated by more than one thread at a time [24].

Every object (i.e., class instantiation) in Java has a built-in lock, which means it has its own boolean lock variable and its own queue for blocked threads. The primary control of the lock is through *synchronized* blocks created with the `synchronized` keyword as follows:

```
Synchronized (ratingsMatrix)
{
}
```

The code above synchronizes the object `ratingsMatrix`, so that only one thread can access it at a time.

We felt that the monitor synchronization did not provide sufficient synchronization functionality for our CoFE needs. So, in addition to using the built-in language synchronization, we implemented a multiple readers, single writer lock (read-write lock) for synchronization purposes. A read-write lock allows multiple threads to acquire a read lock concurrently, provided no other thread has a write lock on the same object. A thread can acquire a write lock if no other thread owns either a read lock or a write lock. We implemented this because ~~Java does not provide a ready-made solution for read-write locks-we wanted to enable a higher level of concurrent execution.~~

Our strategies for addressing synchronization problems were:

- i. synchronizing access to all shared data structures;
- ii. having all threads acquire locks in the same order;
- iii. holding a lock for as little time as possible.

Strategy i ensured that our execution was correct, strategy ii ensured that no deadlock was possible, and strategy iii ensured that we minimized contention for the locks.

With the efforts of all our group members, we successfully ran a test of 100 threads for 72 hours free from any synchronization problems.

(b) Performance testing

Theoretically, running multiple threads increases performance by avoiding busy waits increasing CPU utilization when other threads read from or write to the database.

However, in the Java language, as Brian Goetz [10] pointed out, “*Because of the rules involving cache flushing and invalidation, a synchronized block in the Java language is generally more expensive than the critical section facilities offered by many platforms, which are usually implemented with an atomic ‘test and set bit’ machine instruction.*” He also asserted that, even when a program contains only a single thread running on a single processor, a synchronized method call is still slower than an unsynchronized method call. “*If the synchronization actually requires contending for the lock, the performance penalty is substantially greater, as there will be several thread switches and system calls required.*”

We found that tThe monitor synchronization mechanism provided by the Java language had a significant-considerable negative impact on the performance of CoFE—adding synchronization to one frequently called method decreased our performance from 153

recommendations/second to 80 recommendations/second with a single thread on a Sun Microsystems Sunfire V800 with 16gigs of RAM.

In order to understand exactly what was causing the performance loss, I implemented a performance test and tried to get the average number of predictions per second over all the 943 users and all the 1682 items of the MovieLens data set. We profiled the system to find what synchronization blocks were taking the most time. Through the performance test, we located several highly expensive synchronization blocks. To increase performance, ~~our strategy was to~~ in addition to trying to shorten the period in which locks were held (monitor locks or otherwise). ~~To this end, we identified key synchronization bottlenecks that involved long accesses where we could~~ use the technique of copyingies of data structures to avoid contention rather than synchronizing. Thus we ~~and~~ sacrificed space to gain speed.

For example, we have a data structure called db\_users, which is a ~~linked~~ list that stores all users' information and is ordered by the users' IDs. db\_users is used by the method loadUsers(), which is responsible for loading users' ratings into memory. When a new user comes in, loadUsers() iterates through the db\_users linked list and inserts the user information according to their ID. Iteration through a linked list takes linear time with respect to the number of items in the list, and thus if the synchronization block includes this entire iteration period, there will be significant performance loss. While one thread is iterating through the list, no other threads can access the list.

We used data structure cloning to shorten the synchronization period in the loadUsers() method. When a user must be loaded into db\_users, we acquire the lock, clone a new linked list from the original db\_users, then immediately release the lock. We then iterate through the cloned db\_users, add new users to that clone. Once the users' ratings have been loaded into the clone, the clone of db\_users becomes the new db\_users. The old db\_users is simply dereferenced, and when no more threads are accessing it, it is garbage collected. In this way, we increase space usage (by cloning), but when the clone time plus

the assignment time is less than the iteration time, we shorten the synchronization ~~time~~ ~~and~~ ~~time~~ and thus make our system faster.

### 2.3 My work on the autotest mode

CoFE serves as a research tool for computer scientists. We are trying to set up a framework so that researchers can easily develop and test their new algorithms, using our standard evaluation procedures, metrics, and data sets.

One way to set up the framework is to have a “factory” file that constructs algorithm ~~objects interface with different~~ based on the name that is names that are passed as an ~~the~~ arguments ~~of to~~ the constructor. A factory is a concept in Object Oriented Programming that simplifies the manipulation of many different implementations of the same interface. However, CoFE contains many different implementations of algorithms – some of which depend on third-party libraries that we cannot or do not want to redistribute. In order to access those algorithms from the factory classe, in order to load an algorithm this way, if the algorithm depends on other third party classes, we must first import all of all of the third party classes of that algorithm them. This is inconvenient because each time we want to compile the factory class with a new algorithm ~~test one specific algorithm only~~, we have to ~~compile the~~ have access to the compiled code of all the other algorithms, along with any classes they depend on, otherwise compile complaints will result.

~~For example, if we only want to test a new SVD-based algorithm, we would like to import only those SVD related third party libraries. However, with the factory file framework, during the compiling process, since other algorithms might also depend on their own third party libraries, we need to also have those third party libraries (e.g., the Bayesian net libraries for the Bayesian based algorithms) in the working directories.~~

A better way to achieve a light-weight algorithm experimental framework is to use dynamic class loading, an ~~important~~ feature of the Java Virtual Machine. Here, classes are loaded on demand. Class loading is delayed until the class is first referenced.

To apply the dynamic class loading in our case, first we compile all algorithms along with their third parties once (necessary since the dynamic class loading works on compiled code). Next, we use `Class.forName("AlgorithmName")` to construct the specific algorithm class with the name of "AlgorithmName". Finally, we use `java.lang.reflect` API to create an instance and invoke the methods of that algorithm.

With dynamic class loading, we no longer ~~need the algorithm factory file, and we no longer need a lot of algorithm dependent code. Moreover, except for the first time compile, we do not need to have the~~ need to import implementation of algorithms ~~algorithms' third party classes for compiling our system into the algorithm factory class.~~

← **Formatted:** Bullets and Numbering

### 3 Implementation and evaluation of an improved recommendation algorithm based on Singular Value Decomposition

#### 3

Many collaborative filtering algorithms attempt to filter objects based on personal taste by grouping similar people together as neighbors. In their paper, Herlocker *et. al.* [12] [13] [14] provided a set of recommendations to guide the design of neighborhood-based prediction systems. They demonstrated that best- $n$  neighbors proved to be the best approach to select neighbors to form a neighborhood. In addition, they found that weighting the contributions of neighbors by their correlation with the user increased the accuracy of the end prediction.

However, the nearest neighbor algorithm has been criticized by many researchers. They claim that the algorithm shows limitations, especially since it cannot deal with problems arising from sparsity of data [2] [21]: ~~For example~~First, correlation between two user profiles can only be computed based on items that both users have rated. If users can choose among thousands of items to rate, it is likely that the overlap of rated items between two users will be small. Therefore, many of the computed correlation coefficients are based on just a few observations, and the computed correlation cannot be regarded as a reliable measure of similarity. Second, current approaches measure whether two user profiles are positively correlated, not correlated at all, or negatively correlated. However, ratings given by one user can still be good predictions for the ratings of another user, even if the two user profiles are not correlated. Consider the case where user A's positive ratings are a perfect predictor for a negative rating from user B. However, user A's negative ratings do not imply a positive rating from user B. In other words, you have an asymmetric causality relationship. In such a case, the correlation between the two profiles would be small, and thus potentially useful information is lost. Third, two users can only be similar if there is overlap among the rated items. If users did not rate any common items, their user profiles cannot be correlated. However, just knowing that users did not rate the same items does not necessarily mean that they are not like-minded. Consider the following example. Users A and B are highly correlated, as are users B and C. This relationship provides information about the similarity between users A and C as

Formatted: Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, ... + Start at: 2 + Alignment: Left + Aligned at: 0" + Tab after: 0.25" + Indent

well. However, in case users A and C did not rate any common items, a correlation-based similarity measure could not detect any relation between the two users [2].

### 3.1 Advantages of dimensionality-reducing algorithms

The weakness of Pearson nearest neighbor algorithm for large, sparse databases led us to explore alternative collaborative filtering algorithms. Researchers have been searching for methods for condensing the information contained in the original large, sparse dataset into a smaller set of dimensions with a minimum loss of information. Methods known to our group include: Singular Value Decomposition (SVD) [2] [21] [1] [20] [22], Principal Component Analysis (PCA) [11], and Factor Analysis (FA) [4].

From a literature review, we can summarize the advantage of these dimensionality-reducing algorithms from different perspectives:

- 1) *Dimensionality-reducing algorithms can capture the latent association between different products.*

In real life scenarios, different product names can refer to the similar objects. Correlation based recommendation systems are unable to find this latent association and thus treat these products differently. For example, let us consider two customers—one of them rates 10 different *recycled letter pad* products as “high” and another who rates 10 different *recycled memo pad* products “high.” Correlation based recommendation systems would see no match between these two products and would be unable to discover the latent association that both of them like recycled office products [21] [20].

- 2) *Dimensionality-reducing algorithms can discard noisy information in the original data.*

The rating matrix with reduced dimensions captures most of the important underlying structure in the association of items and users, yet, at the same time, removes noise or variability from the ratings. Berry *et. al.* (1995) [1] analyzed the Latent Semantic

Indexing (LSI) approach in the text retrieval field using SVD. Since LSI does not depend on literal keyword matching, it is especially useful when the text input is noisy, as in OCR (Optical Character Reader). If there are scanning errors and a word (*Dumais*) is misspelled (as *Duniais*), and if these incorrectly spelled words also occur in a context that contained a correctly spelled version of *Dumais*, then *Dumais* will probably be near *Duniais* in the reduced dimensional space. Likewise, the dimensionality-reducing collaborative filtering approach can also discard information that people do not consider informative for their classification task and remove out of scale data that can make the valuable data pop out [1].

- 3) *Dimensionality-reducing algorithms can make some machine-learning algorithms more efficient.*

For a large database containing many users, we may end up with thousands of features while the amount of training data remains highly limited. For a machine-learning algorithm, learning under these conditions is not practical, as the number of data points needed to approximate a concept in  $d$  dimensions grows exponentially with  $d$ , thus making the learning work very expensive [2]. For those algorithms, dimensionality-reducing algorithms can make them more efficient.

### 3.2 Linear algebra of SVD

In this project, we focused on the most popular dimensionality-reducing algorithm—SVD. The linear algebra of SVD can be summarized as follows [1]:

Given an  $m \times n$  matrix  $A$ , where without loss of generality  $m \geq n$  and  $\text{rank}(A) = r$ , the SVD of  $A$ , denoted by  $\text{SVD}(A)$ , is defined as

$$A = U\Sigma V^T,$$

where  $U^T U = V^T V = I_n$  and  $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_n)$ ,  $\sigma_i > 0$  for  $1 \leq i \leq r$ ,  $\sigma_j = 0$  for  $j \geq r + 1$ . The first  $r$  columns of the orthogonal matrices  $U$  and  $V$  define the orthonormal eigenvectors associated with the  $r$  nonzero eigenvalues of  $AA^T$  and  $A^T A$ ,

respectively. The columns of  $U$  and  $V$  are referred to as the left and right singular vectors, respectively, and the singular values of  $A$  are defined as the diagonal elements of  $\Sigma$ , which are the nonnegative square roots of the  $n$  eigenvalues of  $AA^T$ .

It can be shown in linear algebra that the reconstructed matrix  $A_k = U_k \Sigma_k V_k^T$  provides the best lower rank approximations of the original matrix  $A$ , in terms of the Frobenius norm (square-root of sum of squares of matrix elements) [1]. The Frobenius norm measures the variance in the variables; the variance captures information about the data. Thus, when the reduced matrix  $A_k$  is the best approximation of the original matrix  $A$  in terms of the Frobenius norm, we say it captures the most information about the data in  $A$ .

The SVD representation provides a ~~good~~ ~~ideal~~ framework for dimensionality reduction, because one can now quantify the amount of information that is lost if singular values and their corresponding singular vector elements are discarded (through measuring how much Frobenius norm is reduced). The smallest singular values and their corresponding singular vectors are discarded, reducing the dimensionality of the new data representation. The underlying intuition is that the  $k$  largest singular values together with their corresponding singular vector elements capture the important “latent” structure of the original matrix. In so doing, random fluctuations are eliminated.

### 3.3 Application of SVD technique to movie recommendations

#### 3.3.1 SVD decomposition

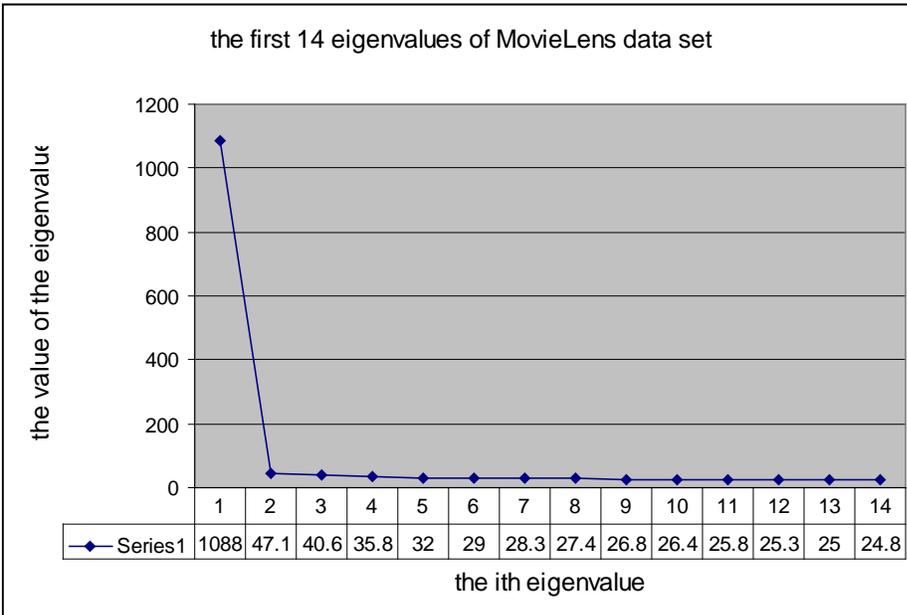
For the SVD decomposition, we utilized the Java Numerical Library (JNL), implemented by the Visual Numerics Company. The original EachMovie data set downloaded from <http://www.grouplens.org> is a  $100000 \times 4$  matrix, composed of 943 users and 1682 items.

To implement the SVD decomposition, we had to convert it into a  $943 \times 1682$  sparse matrix. Also, we transformed the user-item matrix into a normalized rating matrix by (a)

filling the missing cells in the matrix with item (column) averages and ~~then~~ (b) subtracting the user’s average rating from each of their ratings.

Figure 1 presents the resultant eigenvalues (data set: MovieLens data, number of ratings = 80,000).

**Figure 1: The first 14 eigenvalues of the MovieLens data set**



From Figure 1, we can see that the singular values in  $\Sigma$  drop fairly quickly. In other words, the latent feature described in the  $\Sigma$  matrix by 1088.0 is a fairly important feature. However, we do not know what this feature is (perhaps some quality of the movies, some feature of the episodes, or something inexplicable). We will use 14 as the number of eigenvalues in the rest of our experiments. The reason we have selected 14 is that Sarwar *et al.* (2000) [21] use this number in their paper, and we would like to follow their procedures exactly and compare results. By finding a user's reaction to only the most important features, we can generate predictions for the rest of the items.

### 3.3.2 Application 1: Directly use the truncated rating matrix as a prediction matrix (Sarwar *et. al.* (2000) [21])

Using SVD, the  $n \times m$  user-item rating matrix can be factored into three matrices as follows:

$$A = U \cdot S \cdot V'$$

~~Sarwar *et. al.* (2000) [21] claim that the matrices obtained by performing this SVD operation are particularly useful because SVD provides the best lower rank approximations of the original matrix  $A$ , in terms of the Frobenius norm. In addition, it is possible to reduce the  $r \times r$  matrix  $S$  to have only the  $k$  largest diagonal values and thereby obtain a matrix  $S_k, k < r$ . If the matrices  $U$  and  $V$  are reduced accordingly, then the truncated matrix  $A_k = U_k \cdot S_k \cdot V_k'$  is the closest rank- $k$  matrix to  $A$ . The truncated matrix  $A_k$  can capture the meaningful latent relationships between users and items, thus allowing us to compute the predicted rating of certain item by a user.~~

We implemented this algorithm as follows:

- A. Normalized the rating matrix by first filling in missing cells with column averages and then subtracting row averages;
- B. Called methods in the JNL library to apply SVD decomposition;
- C. Truncated the three resulting matrices  $U, S, V$  ( $V'$  is the transpose of  $V$ ), leaving only  $k$  columns in  $U$ ,  $k$  rows in  $V$ , and  $k \times k$  rows and columns in  $S$ , where  $k$  is 14 ;
- D. Constructed the new rating matrix  $A_k = U_k \cdot S_k \cdot V_k'$  as the prediction matrix;
- E. Compared  $A_k$  with the test set in the original rating matrix and computed MAE.

### 3.3.3 Application 2: Use SVD for neighborhood selections (Pryor (1998) [20])

Instead of finding similar users by directly comparing users' ratings for each item, this algorithm selects a neighborhood by comparing users' responses to the set of "features" that make up those items. Pryor (1998) [20] claimed that by transforming into the feature space, this algorithm can eliminate the error that occurs when two users are highly

correlated and yet accurate ratings cannot be given because these two users have no co-rated items.

After SVD, the user-item rating matrix can be factored into  $\underline{U} \cdot S \cdot V'$ .  $\underline{U}$  is representative of the response of each user to certain features.  $V'$  is representative of the amount of each feature present in each item.  $S$  is a matrix related to the feature importance in the overall determination of the rating. Taking  $\underline{U}$ , and multiplying by  $S$ , we move users into a feature space. Now, by simply using normal correlation, we may be able to find a better set-neighborhood of similar users.

1) **For SVD+Cosine**

We implemented this algorithm as follows:

- A. Filled the missing cells in the rating matrix with column averages;
- B. Applied SVD decomposition;
- C. Truncated the three SVD component matrices, resulting in  $U_k$ ,  $S_k$ , and  $V_k$ , where  $k$  is 14;
- D. Multiplied  $U_k$  with  $S_k$ , forming a new user-feature matrix, which will be used as the basis for computing user similarities;
- E. Computed the cosine distance (proportional to the angle between vectors in multidimensional space) between two users using the new matrix  $\underline{U}_k \cdot S_k$  in the following manner. We first calculated the norm (sum of squares of ratings) of each user and then computed  $(U_k S_k) \cdot (U_k S_k)^T$ , which gave us a matrix, whose elements are the sum of all products between the features of two users, and then, finally, we divided the elements in this matrix with the norms of the corresponding users;
- F. Computed the prediction for user  $i$ 's rating of item  $j$ :

$$\frac{\sum_{k=1}^N \text{Cos}(i,k) * \text{rating}(k, j)}{\sum_{k=1}^N \text{Cos}(i,k)}$$

where  $N$  is 943, the total number of users in MovieLens data set, and  $\text{Cos}(i,k)$  is the Cosine distance between user  $i$  and user  $k$ , and  $\text{rating}(k, j)$  is user  $k$ 's rating of item  $j$ ;

**G.** Compared our predicted ratings with the test matrix and computed MAE.

## 2) For SVD+Pearson

We implemented this algorithm as follows:

- A.** Filled the missing cells in the rating matrix with column averages;
- B.** Applied SVD decomposition;
- C.** Truncated the three component matrices, resulting in  $U_k$ ,  $S_k$ , and  $V_k$ , where  $k$  is 14 for now;
- D.** Multiplied  $U_k$  with  $S_k$ , forming a new user-feature matrix, which will be used as to compute user similarities;
- E.** Computed the Pearson correlation between two users using the new matrix  $U_k \cdot S_k$ . We first computed the standard deviations of each user and then subtracted the mean of each user from each element in the matrix  $U_k \cdot S_k$ . Next, we computed  $(U_k S_k) \cdot (U_k S_k)^T$  using the new  $U_k S_k$  matrix. Finally, we divided each element in the  $(U_k S_k) \cdot (U_k S_k)^T$  by the standard deviation of the corresponding user, getting the Pearson correlations between each user pair;
- F.** Sorted the Pearson correlations in reversed order;
- G.** Computed the prediction for user  $i$ 's rating of item  $j$ :

$$\text{meanRating}(i) + \frac{\sum_{k=1}^K \text{similarityWeight}(i,k) * (\text{Rating}(k, j) - \text{meanRating}(k))}{\sum_{k=1}^K \text{similarityWeight}(i,k)}$$

where,  $K$  is the number of neighbors for user  $i$ , and  $similarityWeight(i,k)$  is the Pearson correlation between user  $i$  and his/her neighbor  $k$ . We used  $K = 30$  for now. We compared our predicted ratings with the test matrix and computed MAE.

### 3.4 Evaluation of SVD based algorithms

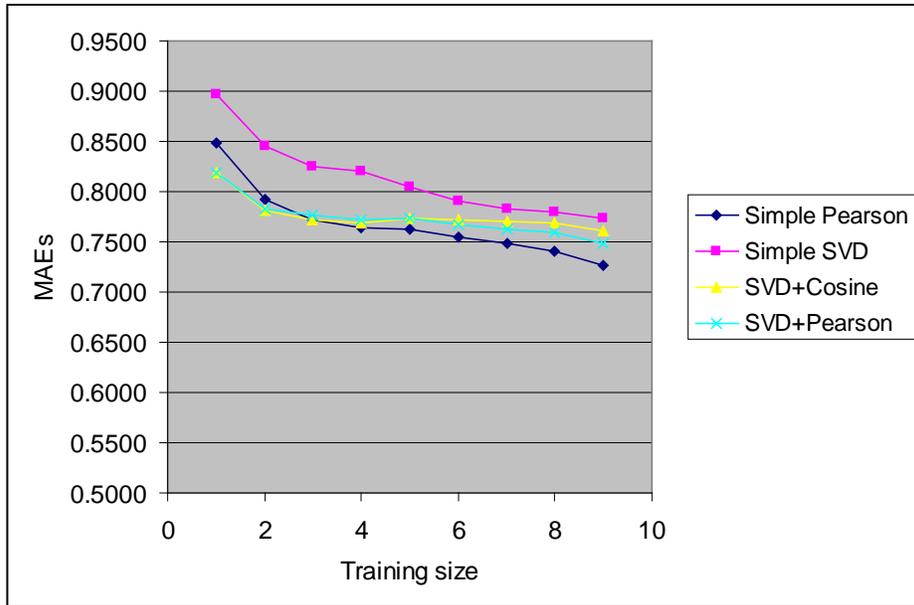
We used Mean Absolute Error (MAE) as our evaluation metric. Table 1 is a summary of the MAEs from the Simple Pearson, Simple SVD, SVD+Cosine, and SVD+Pearson algorithms.

**Table 1: MAEs of Simple Pearson, Simple SVD, and SVD+neighborhood algorithms. The best score in each row is in bold. When the size of the test set is small, SVD+Cosine performs best. With larger amounts of test data, Simple Pearson performs best.**

Training/test splits	Simple Pearson	Simple SVD	SVD+Cosine	SVD+Pearson
0.1/0.9	0.848403786	0.896682863	<b>0.818241909</b>	0.818707273
0.2/0.8	0.791410770	0.844544593	<b>0.780829880</b>	0.782752258
0.3/0.7	0.772564220	0.825732633	<b>0.772275668</b>	0.776148866
0.4/0.6	<b>0.764261045</b>	0.820353091	0.769497278	0.771828499
0.5/0.5	<b>0.762864428</b>	0.803944107	0.773496781	0.773972013
0.6/0.4	<b>0.754859059</b>	0.790581567	0.772187676	0.767435073
0.7/0.3	<b>0.748268140</b>	0.782644821	0.770826229	0.762660002
0.8/0.2	<b>0.740200000</b>	0.779960000	0.769016408	0.759352629
0.9/0.1	<b>0.726177330</b>	0.772820000	0.761434185	0.749216019

Figure 2 was drawn using the values from Table 1.

**Figure 2: MAE comparison between Simple Pearson and SVD-based algorithms**



From Figure 2, we observe that:

- A. Simple SVD has the worst performance—it always lies above other curves. This result is not consistent with that of Sarwar *et al.* (2000) [21]. We used the same data set, followed the same procedures, and evaluated the results using the same metric. The only difference may have been in the random splitting of the training and test sets. However, we are not sure if this caused the difference.
- B. When the training/test splits are less than 0.3, SVD+nearest neighbor algorithms perform better than Simple Pearson nearest neighbor. When the training/test splits are bigger than 0.3, SVD+nearest neighbor algorithms perform worse than Simple Pearson nearest neighbor. This observation is in line with that of Sarwar *et al.* (2000) [21], although they drew it from a comparison between the Simple Pearson nearest neighbor and Simple SVD.

Our results confirm our prior belief that the Simple Pearson algorithm is susceptible to data sparsity since the neighborhood formation process is hindered by the lack of enough training data. On the other hand, SVD-based prediction algorithms can overcome the sparsity problem by utilizing latent relationships. However, as the training data is increased, the accuracy of both SVD-based algorithms and the Simple Pearson algorithm improve, but the improvement in the case of the Simple Pearson algorithm surpasses that of the SVD-based algorithms.

- C. There is no significant difference at the 0.05 significance level (t statistic = 0.37,  $p = 0.74$ , for two tail t-test) between the SVD+Cosine and SVD+Pearson algorithms. SVD+Pearson performs a little better when training/test splits are bigger than 0.5. This may be because the Pearson is essentially Cosine but with the data normalized first (i.e., subtracting mean then dividing by standard deviations).

#### *Complexity analysis—how fast is the prediction*

We used JNL (Java Numeric Library) for SVD decomposition in our experiments. Checking the documentation of this library, we found that the SVD routine for this library utilizes the LINPACK linear algebra package [25]. The computational complexity of the SVD method in LINPACK is  $O(n \times m \times m)$  if  $m$  is less than  $n$  or  $O(n \times n \times m)$  if  $n$  is less than  $m$ , where  $n$  is the number of users, and  $m$  is the number of items.

In our experiments, we observed the time of the SVD decomposition is 725 seconds on a Sun Microsystems Blade running Solaris/SunOS 5.8 with 768MB of RAM for a rating matrix of 943 users and 1682 items (no empty cells in the matrix). This result is obtained when the Java maximum heap is set to 300 megabytes<sup>1</sup>.

---

<sup>1</sup> 300MB is a heap maximum. Once close to 300MB are allocated on Java's heap, garbage collection is invoked.

With the data split into a training set of 80,000 ratings, and a test set of 20,000 ratings, the times of our implementations (all on the Sun Microsystems Blade) were:

Simple SVD: 740 seconds (1.85 times Simple Pearson)  
SVD+Cosine: 753 seconds (1.88 times Simple Pearson)  
SVD+Pearson: 1165 seconds (2.91 times Simple Pearson)  
Simple Pearson: 400 seconds

### 3.5 Conclusions and future work

The results from evaluating our implementations of SVD+nearest neighbor algorithms match our expectation. The data suggests that SVD might be a good way to deal with the sparsity problems that plague collaborative filtering. Extracting latent “features” seems potentially helpful. It also seems to solve the problem of two users who are like-minded, but lack the co-rated items needed to compute their correlations. Moreover, this technique can discard noisy information in the original data.

In terms of computational complexity, we can see that SVD takes a very long time ~~e-and also that the time will increase~~ as the numbers of users and items increase. However, if we can move the SVD decomposition offline, then, it would be possible to maintain a reasonable online speed ~~(although still limited for large numbers of users or items)~~. However, further experiments are needed in order to measure how the offline computation affects the speed and accuracy.

In sum, we think SVD is a promising algorithm for collaborative filtering, but one that needs further investigation.

#### 4 Design and implementation of a new recommendation algorithm that applies to iTunes play count data

As Fries *et al.* write in their book, *The MP3 and Internet Audio Handbook—Your Guide to the Digital Music Revolution* [8]: “A revolution is underway in the music industry that is having a greater impact than when compact discs displaced vinyl records in the early 1980s. This revolution is driven by improvements in technologies for compressing digital audio that make it practical to distribute music over the Internet and turn personal computers into digital jukeboxes that can hold thousands of songs. It is forecast that in five years time 90% of music will be sold digitally.”

The digital music revolution helps people share their musical tastes. As music can be more freely uploaded and downloaded over the web, people are exchanging their music files faster and online musical communities are being formed.

Researchers have seen this as a new domain for collaborative filtering. Instead of directly analyzing the acoustic content of a user’s music, it is now possible to use other users’ listening information to acknowledge, understand, and classify a user’s music taste.

However, in this instance, instead of using the explicit rating data, we can use the play count data contained in iTunes and iPod, which are automatically recorded when users listen to music. Compared to the explicit rating data, play count data are more convenient to collect and possibly more objective. For example, Hill *et al.* asked users to re-rate a set of movies they had rated six weeks earlier and found that the Pearson  $r$  correlation between ratings was 0.83 [15]. This indicates that people can’t even reliably predict what they themselves would rate a movie a short time into the future. Breese *et al.* noted that users are more likely to rate items ~~that they like or which that~~ the system presents. This means that recommendation systems influence the actual ratings users give [3]. Thus, if we measure what people do, rather than what they say, we might get a more accurate indication of their tastes and needs.

In the following sections, I will discuss my analysis of the play count data, the challenges in applying collaborative filtering to it, some attempts to solve these problems, my initial proposal of the Top N recommendation algorithm, and a few recommendations for future work.

#### 4.1 Analysis of the play count data

At the time of this report, I had only eight iTunes musical libraries from iTunes users. From these eight libraries, I conducted some initial statistical analysis on the play count data. The following is a table of their descriptive statistics.

**Table 2: Summaries of the eight iTunes users' descriptive statistics (of their playcounts)**

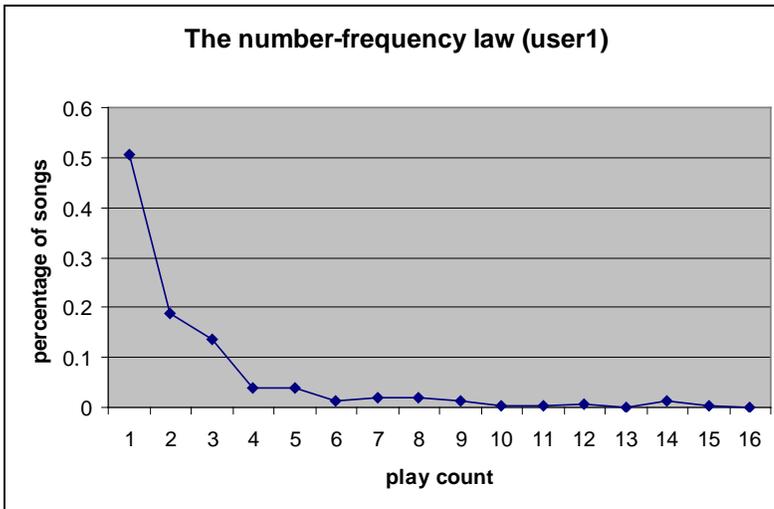
	user1	user2	user3	user4	user5	user6	user7	user8
minimum	1	0	0	0	0	0	0	0
maximum	6	69	362	6	362	15	4	8
mean	1.71	2.82	10.15	0.21	10.02	1.50	0.38	1.17
standard deviation	1.33	5.50	26.16	0.71	25.54	2.55	0.62	1.59

We can see that, in contrast to the explicit rating scale of MovieLens (1,2,3,4,5), which has a restricted scale, the play count data are **bounded only by the lifetime of a user unbounded**. -We can imagine that for an old user, who has been in the system for a long time, and who loves a particular song very much, his play count record for that song would be a very big number. Also, we can notice that for different users the play counts fall in different ranges.

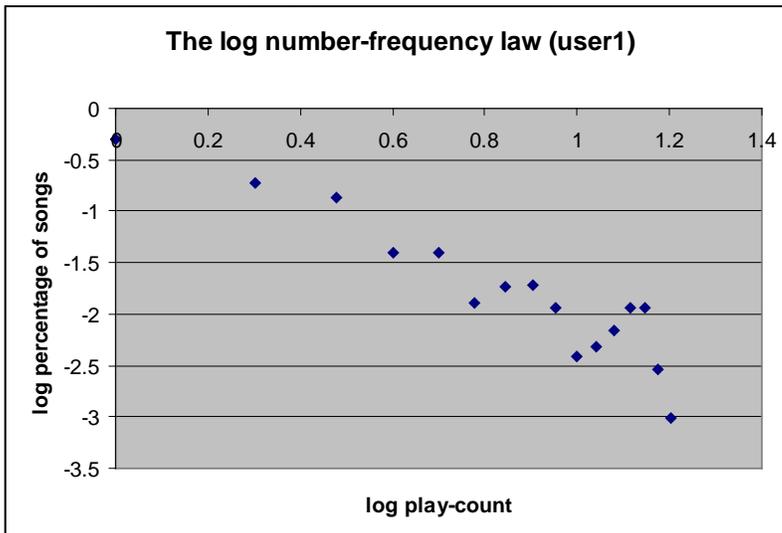
I further analyzed the data to find out if there were any patterns in the play count data. According to the analysis of the eight libraries from real users, I found that the play count data are typical frequency data. The two “Zipf laws” which describe the properties of word frequencies in text documents also hold in our play count data.

- *The number-frequency law.* With  $n$  being the frequency of a word, the plot of  $\log(n)$  versus  $\log(\text{number of words with frequency } n)$  approximates a straight line. This law is observed for the low frequency songs.

**Figure 3: The number-frequency law for User1**

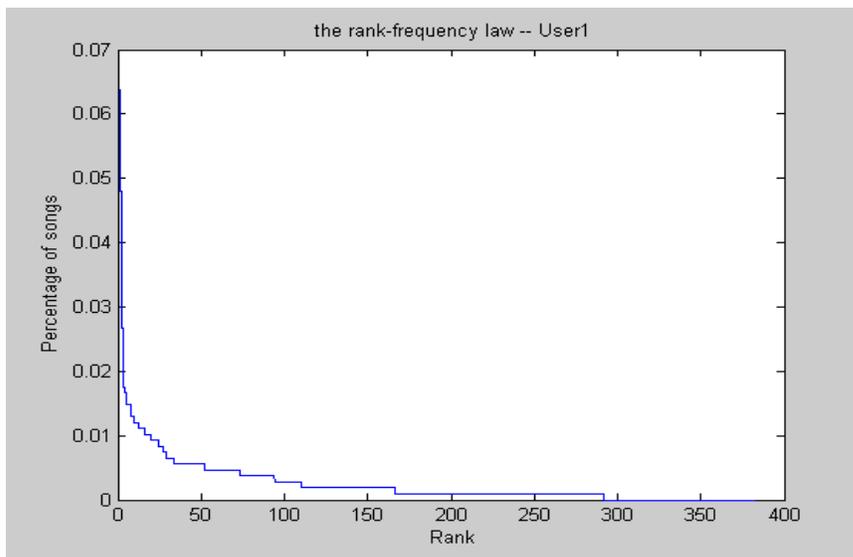


**Figure 4: The log-log plot of the number-frequency law for User1**

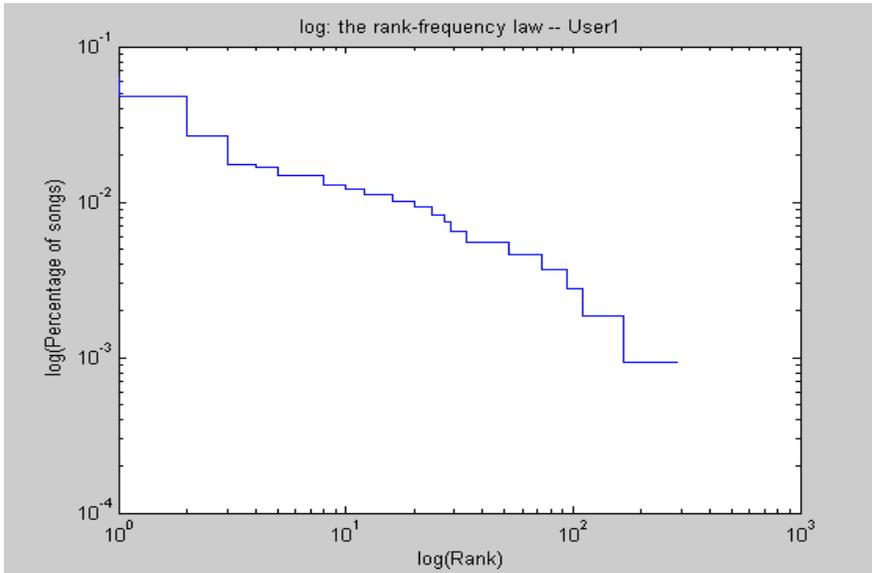


- *The rank-frequency law.* The plot of  $\log(\text{frequency})$  versus  $\log(\text{rank})$  approximates a straight line. This law is observed for the high frequency songs.

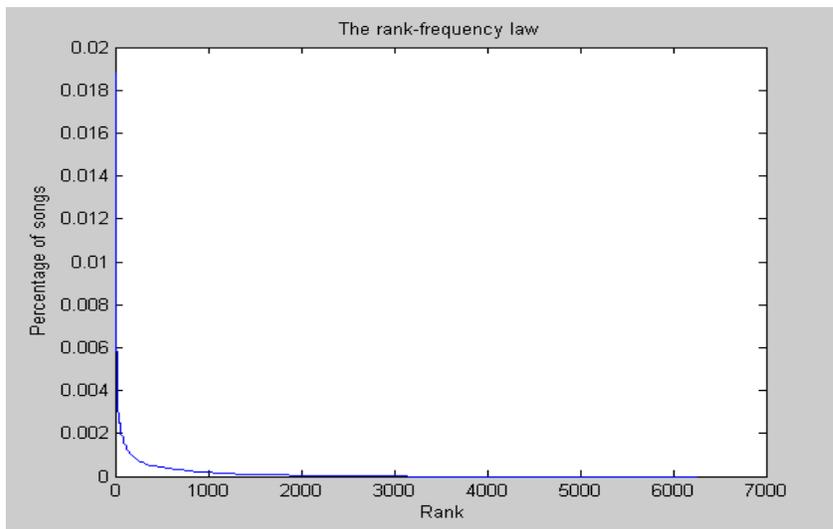
**Figure 5: The rank-frequency law for User1**



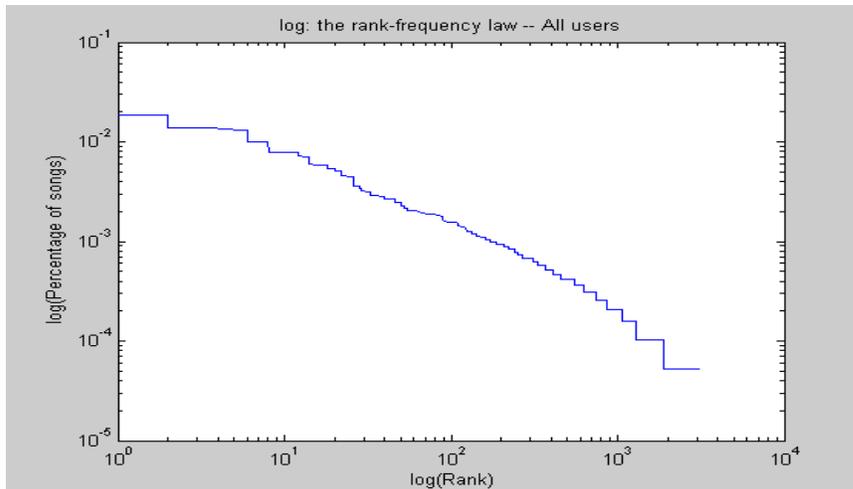
**Figure 6: The log-log plot of the rank-frequency law for User1**



**Figure 7: The rank-frequency law for all users**



**Figure 8: The log-log plot of the rank-frequency law for all users**



## 4.2 Challenges in recommendation algorithms and an ideal algorithm

### 4.2.1 Challenges in applying collaborative filtering to the play count data

The advantage of play count data is that they can be automatically recorded. Compared to the explicitly solicited rating data, collecting play count data has a lower cost to the end users' attention. Also, since play count data are collected automatically every time users listen to music, we can collect more data and the sparsity problem can be alleviated to some extent. Furthermore, as previously indicated, some researchers think the automatically recorded play count data more objectively represent users' preferences.

However, there are challenges in applying traditional collaborative filtering to the play count data.

- (a) Play count data are updated frequently

Every time a user listens to a song, her/his play counts change. This is unlike the explicit ratings in the movie domain, which are generally stable once the user rates the item.

The traditional user-based nearest-neighbor collaborative filtering is computationally expensive when you have many users—it has to compute similarities for each user pair. Also, as pointed out in [17], [18], and [16], traditional recommendation systems do little offline computation because “the similarity between the users is often dynamic in nature, an offline computed user neighborhood can potentially lead to inaccurate prediction scores” [18]. Since the user similarity is computed based on the Cosine or Pearson correlation of two users’ rating vectors, if a user’s ratings change, we have to re-compute the user’s similarity neighborhood in order to have the most up-to-date and best neighborhood. Thus, an offline pre-computed user neighborhood, which cannot reflect changes in dynamic users ratings, would be more inaccurate.

~~Play count data are even more dynamic in nature. Thus, in order to ensure that a system has the most up to date best neighborhood, the online similarity computation would be even heavier.~~

(b) Play counts are unbounded

From the data analysis in the last section, we know that play counts are unbounded. A user’s maximum play count depends on not only how much he/she likes the music, but also on how long the user has been in the system. Does a play count of 5 mean the user likes the song or not? If the user has been just been in the system for just one day, then 5 means-might mean he/she likes the song very much. If they have been in the system two years, it’s a different story altogether. Also, how many songs the user owns can also make a difference.

Therefore, absolute play count value may not be a good representation of preferences. One immediate solution would be to normalize the data, that is, to transform the absolute play count into relative data that is comparable across users. Usually, we can use the following three methods to normalize the data.

(i) Divide the play counts with the user's maximum play count

The intuition behind this method is that we can restrict the users' preference within 0 to 1, where 1 is for the maximum play count and the users' favorite song.

(ii) Divide the play counts with the sum of the play counts of that user

The intuition behind this method is that if we think of the sum of the play counts as the total time the user spends on listening music, the values from this method are the proportions of time the user spends on each song.

(iii) Use standard normalization  $[(\text{playcount} - \text{user's mean})/(\text{users standard deviation})]$

This is the usual way, in statistics, to transform a normal distribution into a standard normal distribution. However, the play count data do not have normal distribution (they are more like an exponential distribution). Thus, this method could be intrinsically flawed.

Approach iii also incorrectly penalizes those users who have more diverse play counts.

For example:

	Song1	Song2	Song3	Song4	mean	std	value
User1	2	2	2	3	2.25	0.5	1.5
User2	1	2	3	10	4	4.08	1.47

From this tiny dataset, we would like to infer that User2 likes Song4 more than User1 does. However, this [normalization](#) method results in a higher value of Song4 for User1, since  $(3-2.25)/0.5 = 1.5 > 1.47 = (10-4)/4.08$ .

The standard deviation is bigger when the user has more diverse play counts. This method makes those users' normalized values lower, which contradicts our intuition that this kind of users should have more relative preference for those songs with higher play counts. In the above example, User2 should have more

preference for Song4 relative to Song1, while User1 should have less preference for Song4 relative to Song1.

Another way to solve the normalization problem is to categorize the play count. No matter how much the maximum play count is, no matter how much total time the user allocates for listening music, and no matter how many deviations of the song away from the mean, our goal is to recommend songs that users might like to listen to. If a user might like to listen to a song 10 times, then this song is worthy of recommendation. Therefore, we could simply categorize the play counts into several categories: for example 1 to 5 could mean not worth recommending; 5 to 10 a mild recommendation; 10 and more a strong recommendation.

A further thought about this normalization problem makes me re-consider the necessity of normalization. Recall that Cosine is invariant to the first two normalization methods, and Pearson correlation is invariant to all three methods **[see appendix]** The Cosine and Pearson correlations are the same before and after some of the normalizations. Therefore, for some algorithms, we do not need to consider the normalization problem.

- (c) The recommendation algorithm still needs to be personalized

We know that traditional user-based nearest neighbor algorithms have several problems when applied to play count data. However, they still have two very important features, which we need to preserve when designing our new algorithms.

- (i) Personalization

A user would like to see that the recommendation is only for her/him, not a popular song that everybody likes.

- (ii) Trust community

As illustrated by the real-world example in which people prefer to seek advice on items of interest (such as a new restaurant) from a friend rather than from a random pedestrian on the street, “trust” plays an important role in users’ feelings about a recommendation. If the user believes that a recommendation comes from someone who is “trustable,” he might like the recommendation better. From our experience, we generally do not like to receive recommendations from people who are not our “type”; we have greater trust in the opinions of people with similar tastes. We would like to have a sense of musical community (neighborhood) and are more likely to consider recommendations that come from that community.

Therefore, our new algorithm should be personalized.

#### 4.2.2 The Ideal Top N recommendation algorithm

In order to meet the challenges of applying collaborative filtering to play count data, an ideal top N recommendation algorithm should have the following features:

- (a) It should be accurate enough to recommend the music that users are most likely to like—this is a requirement for every good top N recommendation algorithm;
- (b) It should be stable enough so that we don’t need to do large re-computation every time the play counts change. Since play count data frequently change, this requirement becomes especially important;
- (c) It should be personal enough so that users can see their recommendations change ~~every time~~as their play counts change. Users would like to see that the changes in their own play counts really affect the system’s recommendation results. Users do not want to see that the recommendations are just popular songs. They want to see that the recommendations are tailored for themselves particularly. Moreover, they also want to know that the recommendations come from users they trust;

(d) The algorithm should be intuitive enough so that we can explain the algorithm to users. An algorithm that is easy to understand can help users to trust the algorithm and the system more and thus use the system more.

### 4.3 An initial proposal on the recommendation algorithms

My initial proposal for the recommendation algorithm was an item-based nearest neighbor algorithm, which is similar to that of Karypis [17]. However, I did some modification, so that this algorithm is more applicable to non-binary data.

#### 4.3.1 Karypis' [17] algorithm

Karypis' [17] algorithm is based on the observation that a user is interested in consuming items that are similar to the items the user liked earlier and will avoid items that are similar to the items the user didn't like earlier.

Karypis' [17] algorithm can be abstracted at high-level into two steps: (a) compute the similarities between different items; and (b) apply these pre-computed similarities to derive the top N recommendations for an active user.

The details of the algorithms are:

Step 1: Compute similarities

The input to this step is the  $n \times m$  user-item rating matrix  $R$ , which can be considered as  $m$  item vectors (columns), where each item vector consists of  $n$  users' ratings for this item. Note that Karypis [17] only applies this algorithm to binary rating data, so each element in matrix  $R$  is either 0 (not purchased) or 1 (purchased). The output is a  $m \times m$  similarity matrix  $M$ , whose elements are the pair-wise similarities between each item.

Karypis [17] proposed two methods for computing the item similarities: Cosine-Based Similarity and Conditional Probability-Based Similarity. Karypis' [17] experimental results did not show the significant superiority of one method over the other.

Step 2: Derive top N recommendations

The input to this step is the similarity matrix  $M$ , a  $m \times 1$  vector  $U$  that stores the  $k$  items that have already been rated (or purchased or listened) by the active user and the number of items to be recommended  $N$ . The active user's information in vector  $U$  is encoded by  $U_i = 1$  if the user has purchased the  $i$ th item and zero otherwise. The output of the algorithm is a  $m \times 1$  vector  $x$  whose non-zero entries correspond to the top  $N$  items that were recommended.

The output vector  $x$  is computed in three steps. First, for each item, sum its similarities with the  $k$  items that the active user has purchased; second, for each item, if it belongs to the  $k$  items that the user has purchased, then zero it out in vector  $x$ ; thirdly, the algorithm sets to zero all the entries of  $x$  that have a value smaller than the  $N$  largest values of  $x$ .

Karypis [17] applied this algorithm to eight data sets, including the MovieLens data sets I used in evaluating the SVD algorithm. However, all the rating sets of all these data sets are binary—they also transformed the 5-scaled MovieLens data set into a binary set.

Karypis [17] claimed that their Cosine-based algorithm showed better accuracy and performance on three out of five of the data sets they tested.

As pointed out in [17], [18], and [16], the key to this algorithm is that it creates the expensive similar-items table offline and thus increases the online speed. In contrast to user similarities, in some cases, the similarities between items are considered relatively more stable because if item similarities are computed using more ratings than those used in computing user similarities (i.e., the co-rated item ratings are more than the co-rated

user ratings), they will be less affected by changes in ratings. Although moving some computation offline would make the recommendation less up-to-date with changes in ratings and would thus affect accuracy, if the item similarities are relatively stable when we move their computation offline, the accuracy of the recommendation is not greatly affected. From this perspective, the item-based algorithm might be faster.

#### 4.3.2 My modification

Play count data are updated frequently. Every time a user finishes listening to a song, her/his play counts change. This is one of the differences between play count data and explicit rating data. For example, in the MovieLens rating data, once the user rates an item, that rating is unlikely to change.

As discussed above, item-based algorithms such as Karypis' [17] can be more scalable and efficient. They compute the expensive item similarities offline (based on the assumption that item similarities are relatively stable), and thus is especially good for the iTunes play count data, which are more dynamic.

However, it was not Karypis' [17] original intention to apply the algorithm to a non-binary data set. If we still apply their algorithm to our play count data, then some problems might arise.

One of these problems lies in the step when we try to derive the top N recommendations after computing the item similarities. In Karypis' [17] algorithm, the scores used to sort the items and decide whether the items are worth recommending is the sum of the similarities of the items with those items that had been bought by the active user. They don't consider that some items the user has listened to might be more important than others (i.e., they might have been played a lot more). If we apply this algorithm directly to the play count data, we cannot reflect the differences in users' play counts in our recommendations.

We illustrate this problem using the following example. Table 3 presents the similarities of item2 and item8 with all other items. We can see that item1 is the most similar item with item2 (their similarity is 0.9) and that item9 is the most similar item with item8 (their similarity is also 0.9). Originally at t1, the active user had listened to item1 once, item5 three times, and item9 twice. At t2, he has listened to item1 nine more times.

**Table 3: The similarities of item2 and item8 with all items and their play counts**

sim-matrix	Item1	item2	item3	item4	Item5	item6	item7	item8	item9
item2	0.9	1	0.7	0.6	0.5	0.4	0.3	0.2	0.1
item8	0.1	0.2	0.3	0.4	0.5	0.6	0.7	1	0.9
play counts at t1	1	0	0	0	3	0	0	0	2
play counts at t2	10	0	0	0	3	0	0	0	2

**Table 4: The similarities of item2 and item8 with all items and their play counts and weights**

sim-matrix	Item1	item2	item3	item4	Item5	item6	item7	item8	item9
item2	0.9	1	0.7	0.6	0.5	0.4	0.3	0.2	0.1
item8	0.1	0.2	0.3	0.4	0.5	0.6	0.7	1	0.9
play counts at t1	1	0	0	0	3	0	0	0	2
play counts at t2	10	0	0	0	3	0	0	0	2
Weight1	0.17	0	0	0	0.5	0	0	0	0.33
Weight2	0.67	0	0	0	0.2	0	0	0	0.13

Directly using Karypis' [17] algorithm, we can see from the table below that the recommendation ranking scores for both item2 and item8 in both t1 and t2 are the same. When the user's play count for item1 increases, there are no changes in our recommendations.

**Table 5: Recommendation scores from two algorithms**

		sum without weights	sum with weights
item2	play count 1	1.5	0.4333333333
	play count 2	1.5	0.7133333333
item8	play count 1	1.5	0.5666666667
	play count 2	1.5	0.2866666667

To modify this algorithm and reflect change in the play counts, for each item that an active user has bought, I assign to it a weight (see the row Weight1 and row Weight2 in Table 4). The weight is the percentage of each item's play-count in the sum of the play-count of the user. This way, when the user changes his/her play-count for an item that they have already listened to before, different recommendations result, as illustrated in the table above.

Consider tables 3 and 4 again. In the above example, using my modified algorithm, in play count 1 the score for item 2 is 0.43 and item8 is 0.56. Thus, item8 is scored higher than item2. After the play count of item1 increases to 10, the score of item2 is 0.71 and item8 is 0.29. Thus, item2 is preferred to item8. The result is reasonable: when the user's play count increases from 1 to 10, we have more evidence that the user is more interested in item1, and thus item2, which is very similar to item1 (similarity with item1 is 0.9), should also be preferred.

This modification has the following advantages:

- (a) We still maintain the item-item algorithm's assumption that the similarities between items are stable, and a user will be more likely to listen to music that is similar to the music that he/she has already listened;
- (b) We can still compute the similarities offline. Therefore, we can still maintain the scalability advantage of Karypis' [17] algorithm;
- (c) By applying this modification, we can reflect user's changes online and can handle non-binary ratings data.

### 4.3.3 A proposal on pre-clustering

According to Karypis [17], a major criticism of item-based algorithms is that they are not personalized, that is recommendations are based on all users' information. Item-based

algorithms are based on the global similarities between items, and the global item similarities are computed using all users' ratings (or play counts). But a user might like to have recommendations based only on those users that have similar tastes. This is different from the user-based algorithm, where we can find a neighborhood of people that have the same musical tastes. Based on this analysis, a pre-clustering before the item-based algorithm might be a good proposal. We will describe what we mean by pre-clustering.

Clustering techniques work by identifying groups of users who appear to have similar preferences. Once the clusters are created, predictions for an individual can be made by averaging the opinions of the other users in that cluster, excluding users who are not part of the cluster. Therefore, clustering techniques can be applied as a "first step" for shrinking the candidate set from the whole data set to only the neighborhood sets. Pre-clustering thus also has the benefit of increasing performance by shrinking the data set.

It is an easy step to apply pre-clustering in our music recommendation, as the clustering algorithm is one of the already implemented algorithms in our CoFE system. We can first cluster the users into several clusters (several "musical communities"), then, inside each cluster, we apply our modified item-based algorithm to derive the top N recommendations.

#### **4.3.4 Summary of my proposed algorithm**

The following is a summary of my proposed top N recommendation algorithm for the play count data.

- (a) Pre-cluster. Cluster the users into several musical communities;
  
- (b) Compute item similarities. Within each user community, compute the similarities between each item pair—simply use Cosine similarity. Normalize the similarities so that the sum of an item's similarities with all items adds up to one;

(c) Derive top N recommendations. Multiply each item's similarities with the items that the active user has bought by those items' play count weights, and compute each item's score. Sort the scores and recommend those items that have the highest scores.

#### **4.3.5 Criticism of the algorithm**

We don't have enough real data to evaluate the accuracy and performance of this algorithm. However, from a theoretical point of view, this algorithm answers some of the challenges we listed before.

First, it has faster online speed. It creates the expensive similar-item table offline and reduces the online computation.

Second, the algorithm's assumptions are reasonable. The reason that the similar-item table could be computed offline is that the similarities between items are relatively stable, unlike the similarities between users, which change every time the play counts change. In addition, the algorithm's intuition that a user is interested in purchasing the items that are similar to the items the user liked earlier is also reasonable.

Third, since Pearson correlations are invariant to normalizations [see **Appendix**], this algorithm is invariant to normalization (we don't normalize weights, since the standard normalization of weights do not exist). We don't need to worry about normalizing the unbounded play count data, as the results from normalized data and un-normalized data are the same.

Fourth, if we add a pre-clustering step, then we can make up for the defect of item-based algorithms, that is, the lack of personalization.

#### **4.4 Summary and future work**

In sum, in this project, my major contributions with respect to the iTunes data include:

(a) Data analysis of iTunes play count data;

(b) Improvement on Karypis' item-item top N recommendation algorithm to support non-binary ratings

Theoretically, the proposed algorithm answers some of the challenges in applying collaborative filtering to iTunes play count data. However, we have difficulties in proving this empirically. So far, I only have eight real user music libraries. We cannot evaluate the algorithm without real data. We need to test whether the songs we recommend to users are really those that the users like, but without real data, we don't know what the users really like. In future work, if we collect more real user libraries, we could run additional experiments and perhaps discover problems in the proposed algorithm and thereby improve it.

Another area that we need to work on in the future is how to utilize the play time and date data. The songs that users listened to recently should have a greater preference over songs listened to in the distant past. How to utilize these data is also a very interesting and challenging problem.

## 5. Summary and future work

In this report, I first introduced three areas that collaborative filtering researchers are interested in and challenged by, namely: (a) how to ~~solve~~address the sparsity and scalability problems in recommendation systems; (b) how to rapidly develop and test collaborative filtering algorithms; and how to synthesize various algorithms; and ~~(c) how~~ to apply collaborative filtering to implicit unbounded numeric preference data. In the next three chapters, I described my contributions to these three areas: (a) my work on the CoFE recommendation engine; (b) my work on the SVD based algorithm; and (c) my attempt to design a new algorithm for the iTunes play count data.

With my and our team members' efforts, the CoFE recommendation engine has become efficient, productive, and freely downloadable software and a research framework that has known to be downloaded 440 times. We have empirically proved that the SVD based algorithm is more accurate for sparse data sets. Currently, we are collecting additional user music libraries to test my proposed algorithm and ultimately create a better algorithm.

In the future, CoFE needs to be given a user-friendly interface, making it a more convenient tool for researchers to develop and evaluate their new algorithms. Second, we need to collect more libraries to test and improve the proposed algorithm. And, finally, we need to utilize the play time and date data to design more accurate recommendation algorithms.

## 6. References

- [1] Berry, M., Dumais, S., and O'Brien, G., *Using Linear Algebra for Intelligent Information Retrieval* SIAM Review, 37, 4, Dec. 1995, pp. 573-595.
- [2] Billsus, D., and Pazzani, M. (1998) *Learning Collaborative Information Filters* in Proceedings of the International Conference on Machine Learning (Madison WI, July 1998), Morgan Kaufmann Publishers.
- [3] Breese, J., Heckerman, D., and Kadie, C., *Empirical Analysis of Predictive Algorithms for Collaborative Filtering* MSR-TR-98-12 May 1998 18 p.
- [4] Canny, J. (2002) *Collaborative Filtering with Privacy via Factor Analysis* SIGIR'02, August 11-15, 2002, Tampere, Finland
- [5] Daniel, L., Godde, O., Maclaughlin M., Nong, S., Wang, Y., and Herlocker, J. (2004) *Challenges and Solutions for Synthesis of Knowledge regarding Collaborative Filtering Algorithms*, manuscript, Oregon State University
- [6] Deshpande, M. and Karypis, G. *Item-based Top-N Recommendation Algorithms* ACM Transactions on Information Systems, 22(1):1-34, 2004
- [7] Ehrenberg, A.S.C. *Repeat-buying: facts, theory and application*. Charles Griffin & Company Ltd. London, 1988
- [8] Fries B., and Fries, M., "The MP3 and Internet Audio Handbook – Your Guide to the Digital Music Revolution". TeamCom Books, February 2000, ISBN 1-928791-10-7
- [9] Godde, O., *A Java-Based Recommendation Engine* Master Project Report
- [10] Goetz, B. "When do we have to synchronize, and how expensive is it really" <http://www-106.ibm.com/developerworks/java/library/j-threads1/>
- [11] Goldberg, K., Roeder, T., Gupta, D., and Perkins, C., *Eigenstate: A Constant Time Collaborative Filtering Algorithm* Information Retrieval 4, 2
- [12] Herlocker, J. (2000) *Understanding and Improving Automated Collaborative Filtering System*. Ph.D dissertation, University of Minnesota
- [13] Herlocker, J., Konstan, J., Borchers, A., and Riedl, J., (1999) *An Algorithm Framework for Performing Collaborative Filtering* Proceedings of the 1999 Conference on Research and Development in Information Retrieval. Aug. 1999.
- [14] Herlocker, J., Konstan, J., Riedl, J., (2002) *An Empirical Analysis of Design Choices in Neighborhood-Based Collaborative Filtering Algorithms* Information Retrieval, 5 (2002), Kluwer Academic Publishers, 287-310.

- [15] Hill, W., Stead, L., Rosenstein, M., and Furnas, G., “*Recommending and Evaluating Choices in a Virtual Community of Use*” Bellcore CHI’95
- [16] Johnson, S., “*So Much information, So Little Relevance*”, ComputerWorld, August, 02, 2004
- [17] Karypis, G. *Evaluation of Item-Based Top-N Recommendation Algorithms* In Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM), Atlanta, GA, 2001. pp. 247-254, 2001
- [18] Linden, G., Smith, B., and York J. *Amazon.com Recommendations: Item-to-Item Collaborative Filtering* IEEE Internet Computing 7, 1 (January /February 2003), 76-80.
- [19] Michell, T., (1997) *Machine Learning* McGraw Hill, 1997
- [20] Pryor, M., (1998) *The Effects of Singular Value Decomposition on Collaborative Filtering* Master thesis, Dartmouth College
- [21] Sarwar, B., Karypis, G., Konstan, J., and Riedl, J., *Application of Dimensionality Reduction in Recommender System – A Case Study* In Proceedings of the WebKDD Workshop at the ACM SIGKDD. ACM, New York. [22] Syu, I., Lang, S., and Deo, N., *Incorporating Latent Semantic Indexing into a Neural Network Model for Information Retrieval* In Proceedings of the fifth international conference on Information and knowledge management [table of contents](#) pp 145-153
- [23] CoFE manual
- [24] [Pete Van Der Linden](#) Java 2, the 4<sup>th</sup> edition, SunSoft Press
- [25] <http://math.nist.gov/javanumerics/jama/>

## 7. Appendix

1. Prove that Cosine is invariant to a constant factor.

Proof:

Suppose we have two data vectors  $X = (x_1, x_2, \dots, x_n)$  and  $Y = (y_1, y_2, \dots, y_n)$ .

$$\text{By definition, } \cos(X, Y) = \frac{X \cdot Y}{\|X\| \cdot \|Y\|} = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}} \cdot \left( \sum_{i=1}^n x_i^2 \neq 0, \sum_{i=1}^n y_i^2 \neq 0 \right)$$

Let  $a$  ( $a > 0$ ) and  $b$  ( $b > 0$ ) be two constant factors. Cosine ( $aX, bY$ ) is computed as follows.

$$\begin{aligned} \cos(aX, bY) &= \frac{aX \cdot bY}{\|aX\| \cdot \|bY\|} \\ &= \frac{\sum_{i=1}^n (ax_i)(by_i)}{\sqrt{\sum_{i=1}^n (ax_i)^2} \sqrt{\sum_{i=1}^n (by_i)^2}} \\ &= \frac{ab \sum_{i=1}^n x_i y_i}{ab \sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}} \\ &= \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}} \\ &= \cos(X, Y) \end{aligned}$$

Therefore, Cosine is invariant to a constant factor.

Q.E.D.

2. Prove that Pearson correlation is invariant to constant factor.

Proof:

Suppose we have two data vectors  $X = (x_1, x_2, \dots, x_n)$  and  $Y = (y_1, y_2, \dots, y_n)$ . Let  $\bar{x}$  and  $\bar{y}$  be the means of  $X$  and  $Y$ , respectively.

By definition, the Pearson correlation of  $X$  and  $Y$  is computed as follows.

$$\text{corr}(X, Y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}} \quad \left( \sum_{i=1}^n (x_i - \bar{x})^2 \neq 0, \sum_{i=1}^n (y_i - \bar{y})^2 \neq 0 \right)$$

Let  $a$  ( $a > 0$ ) and  $b$  ( $b > 0$ ) be two constant factors. The Pearson correlation of  $aX$  and  $bY$  is

$$\begin{aligned} \text{corr}(aX, bY) &= \frac{\sum_{i=1}^n (ax_i - a\bar{x})(by_i - b\bar{y})}{\sqrt{\sum_{i=1}^n (ax_i - a\bar{x})^2 \sum_{i=1}^n (by_i - b\bar{y})^2}} \\ &= \frac{ab \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{ab \sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}} \\ &= \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}} \\ &= \text{corr}(X, Y) \end{aligned}$$

Therefore, Pearson correlation is invariant to constant factor.

Q.E.D.

3. Prove that Pearson correlation is invariant to standard normalization.

Proof:

Suppose we have two data vectors  $X = (x_1, x_2, \dots, x_n)$  and  $Y = (y_1, y_2, \dots, y_n)$ . Let  $\bar{x}$  and  $\bar{y}$  be means of  $X$  and  $Y$ , respectively. Let  $s_X$  and  $s_Y$  be standard deviations of  $X$  and  $Y$ , respectively.

Standard normalization of  $X$  (or  $Y$ ) is defined as  $\frac{X - \bar{x}}{s_X}$  (or  $\frac{Y - \bar{y}}{s_Y}$ ). Note that the means of  $X$  and  $Y$  after standard normalization are 0s.

The Pearson correlation of standard normalized  $X$  and standard normalized  $Y$  is

$$\begin{aligned}
\text{corr}\left(\frac{X - \bar{x}}{s_X}, \frac{Y - \bar{y}}{s_Y}\right) &= \frac{\sum_{i=1}^n \left(\frac{x_i - \bar{x}}{s_X} - 0\right) \left(\frac{y_i - \bar{y}}{s_Y} - 0\right)}{\sqrt{\sum_{i=1}^n \left(\frac{x_i - \bar{x}}{s_X} - 0\right)^2 \sum_{i=1}^n \left(\frac{y_i - \bar{y}}{s_Y} - 0\right)^2}} \\
&= \frac{\sum_{i=1}^n \frac{x_i - \bar{x}}{s_X} \cdot \frac{y_i - \bar{y}}{s_Y}}{\sqrt{\sum_{i=1}^n \left(\frac{x_i - \bar{x}}{s_X}\right)^2 \sum_{i=1}^n \left(\frac{y_i - \bar{y}}{s_Y}\right)^2}} \\
&= \frac{\frac{1}{s_X s_Y} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\frac{1}{s_X s_Y} \sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}} \\
&= \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}} \\
&= \text{corr}(X, Y)
\end{aligned}$$

Therefore, Pearson correlation is invariant to standard normalization. Q.E.D.

4. Prove that weights are invariant to a constant factor.

Proof:

Suppose we have a data vector  $X = (x_1, x_2, \dots, x_n)$ .

By definition, the weight of  $x_i$  is computed as follows.

$$\text{Weight of } x_i = \frac{x_i}{\sum_{j=1}^n x_j} \quad \left( \sum_{j=1}^n x_j \neq 0 \right)$$

Let  $a > 0$  be a constant factor,

$$\text{Weight of } ax_i = \frac{ax_i}{\sum_{j=1}^n (ax_j)} = \frac{ax_i}{a \sum_{j=1}^n x_j} = \frac{x_i}{\sum_{j=1}^n x_j} = \text{weight of } x_i. \quad \text{Q.E.D.}$$

5. Prove that weight of standard normalized data does not exist.

Proof:

Suppose we have a data vector  $X = (x_1, x_2, \dots, x_n)$ . Let  $\bar{x}$  and  $s_x$  be the mean and the standard deviation of  $X$ , respectively.

Standard normalization of  $x_i$  is defined as  $\frac{x_i - \bar{x}}{s_x}$ . Note that the mean of  $X$  after standard normalization is 0.

By definition, the weight of  $x_i$  is computed as follows.

$$\text{Weight of } x_i = \frac{x_i}{\sum_{j=1}^n x_j}.$$

The weight of standard normalization of  $x_i$  is computed as follows.

$$\text{Weight of } \frac{x_i - \bar{x}}{s_x} = \frac{\frac{x_i - \bar{x}}{s_x}}{\sum_{j=1}^n \frac{x_j - \bar{x}}{s_x}}.$$

However, the denominator  $\sum_{j=1}^n \frac{x_j - \bar{x}}{s_x}$  is 0 and therefore, mathematically, the weight of the standard normalized data doesn't exist.