# A Distributed, Architecture-Centric Approach to Computing Accurate Recommendations from Very Large and Sparse Datasets

**Serhiy Morozov**

B.A., Computer Science, Westminster College, 2005
M.S., Computer Science, University of Kansas, 2007

*Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy.*

---

Dr. Hossein Saiedian
Professor and Chairperson

---

Dr. Arvin Agah
Professor

---

Dr. Jerzy Grzymala-Busse
Professor

---

Dr. Bo Luo
Assistant Professor

---

Dr. Saul Stahl
Professor

---

Date Defended

The Dissertation Committee for Serhiy Morozov certifies
that this is the approved version of the following dissertation:

**A Distributed, Architecture-Centric Approach to Computing Accurate Recommendations from Very Large and Sparse Datasets**

---

Dr. Hossein Saiedian
Professor and Chairperson

---

Dr. Arvin Agah
Professor

---

Dr. Jerzy Grzymala-Busse
Professor

---

Dr. Bo Luo
Assistant Professor

---

Dr. Saul Stahl
Professor

---

Date Approved

# Abstract

The use of recommender systems is an emerging trend today, when user behavior information is abundant. There are many large datasets available for analysis because many businesses are interested in future user opinions. Sophisticated algorithms that predict such opinions can simplify decision-making, improve customer satisfaction, and increase sales. However, modern datasets contain millions of records, which represent only a small fraction of all possible data. Furthermore, much of the information in such sparse datasets may be considered irrelevant for making individual recommendations. As a result, there is a demand for a way to make personalized suggestions from large amounts of noisy data.

Current recommender systems are usually all-in-one applications that provide one type of recommendation. Their inflexible architectures prevent detailed examination of recommendation accuracy and its causes. We introduce a novel architecture model that supports scalable, distributed suggestions from multiple independent nodes. Our model consists of two components, the input matrix generation algorithm and multiple platform-independent combination algorithms. A dedicated input generation component provides the necessary data for combination algorithms, reduces their size, and eliminates redundant data processing. Likewise, simple combination algorithms can

produce recommendations from the same input, so we can more easily distinguish between the benefits of a particular combination algorithm and the quality of the data it receives. Such flexible architecture is more conducive for a comprehensive examination of our system.

We believe that a user's future opinion may be inferred from a small amount of data, provided that this data is most relevant. We propose a novel algorithm that generates a more optimal recommender input. Unlike existing approaches, our method sorts the relevant data twice. Doing this is slower, but the quality of the resulting input is considerably better. Furthermore, the modular nature of our approach may improve its performance, especially in the cloud computing context. We implement and validate our proposed model via mathematical modeling, by appealing to statistical theories, and through extensive experiments, data analysis, and empirical studies.

Our empirical study examines the effectiveness of accuracy improvement techniques for collaborative filtering recommender systems. We evaluate our proposed architecture model on the Netflix dataset, a popular (over 130,000 solutions), large (over 100,000,000 records), and extremely sparse (1.1%) collection of movie ratings. The results show that combination algorithm tuning has little effect on recommendation accuracy. However, all algorithms produce better results when supplied with a more relevant input. Our input generation algorithm is the reason for a considerable accuracy improvement.

# Contents

**1 Introduction** 1

1.1 Importance of the Study . . . 3

1.2 Recommendation Accuracy Problem . . . 5

1.3 Research Hypothesis . . . 8

1.4 Conceptual Model of a Recommender System . . . 8

1.5 Recommendation Accuracy Metrics . . . 9

1.6 Source of the Experimental Data . . . 11

1.7 Contributions and Organization . . . 13

1.8 Conclusion . . . 15

**2 An Overview of Recommender Techniques** 16

2.1 Content-Based Recommendations . . . 18

2.1.1 The Limited Content Analysis Problem . . . 18

2.1.2 The Overspecialization Problem . . . 20

2.1.3 The New User Problem . . . 21

2.2 Collaborative Filtering Recommendations . . . 22

2.2.1 The Sparsity Problem . . . 22

2.2.2 The New User Problem . . . . . . . . . . . . . . . . . . . . . . 24
2.2.3 The New Item Problem . . . . . . . . . . . . . . . . . . . . . 26
2.3 Hybrid Recommendations . . . . . . . . . . . . . . . . . . . . . . . 27
2.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**3 Collaborative Filtering Algorithms 31**

3.1 Model-Based Algorithms . . . . . . . . . . . . . . . . . . . . . . . 31
3.1.1 Association Rules Model . . . . . . . . . . . . . . . . . . . . . 32
3.1.2 Probabilistic Models . . . . . . . . . . . . . . . . . . . . . . . 33
3.1.3 Singular Value Decomposition Model . . . . . . . . . . . . . . 35
3.1.4 Neural Network Model . . . . . . . . . . . . . . . . . . . . . . 37
3.2 Memory-Based Algorithms . . . . . . . . . . . . . . . . . . . . . . 40
3.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**4 The Proposed System Architecture 47**

4.1 Distributed Computation with Accessible Data . . . . . . . . . . . . . 48
4.2 Multi-Platform Implementation with Minimal Client Requirements . 50
4.3 Complexity Management with Layers . . . . . . . . . . . . . . . . . 52
4.4 System Adaptability with Strict Interfaces . . . . . . . . . . . . . . 55
4.5 Future Extensibility with Prebuilt Components . . . . . . . . . . . . 57
4.6 System Dataflow Bottlenecks . . . . . . . . . . . . . . . . . . . . . 59
4.6.1 Efficient Communication Utilization . . . . . . . . . . . . . . . 60
4.6.2 Efficient Computation Utilization . . . . . . . . . . . . . . . . 63
4.6.3 Efficient Space Utilization . . . . . . . . . . . . . . . . . . . . 64

4.7 Input Generation System Specification . . . . . . . . . . . . . . . . . . 65

4.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

**5 Improving the Quality of Recommender Input 73**

5.1 The Standard Input Generation Approach . . . . . . . . . . . . . . . . 74

5.2 Two Ways of Establishing Neighbor Influence . . . . . . . . . . . . . . 79

5.3 Candidate Input Generation Algorithms . . . . . . . . . . . . . . . . . 81

5.4 Desired Input Qualities . . . . . . . . . . . . . . . . . . . . . . . . . . 83

5.4.1 Bigger Net Weights Produce More Reliable Results . . . . . . 83

5.4.2 Sorting Before Truncating for Maximum Net Weight . . . . . 86

5.4.3 Global Similarity does not Imply Local Similarity . . . . . . . 86

5.4.4 Second Pass Local Similarities are Higher . . . . . . . . . . . 88

5.5 Similarity Refinement Simulation . . . . . . . . . . . . . . . . . . . . 90

5.6 The Effects of Input Resorting on Estimation Accuracy . . . . . . . . 92

5.7 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

**6 A Novel Input Generation Model 97**

6.1 Formalism Motivation, Scope, and Goals . . . . . . . . . . . . . . . . . 98

6.1.1 Static Model Structure . . . . . . . . . . . . . . . . . . . . . . 99

6.1.2 Data Access Behavior Specification . . . . . . . . . . . . . . . 105

6.1.3 Data Processing Behavior Specification . . . . . . . . . . . . . 106

6.2 Static Model Implementation and Analysis . . . . . . . . . . . . . . . . 112

6.3 Behavioral Model Implementation and Analysis . . . . . . . . . . . . 117

6.3.1 Compatibility with the Rest of the Architecture . . . . . . . . 117

6.3.2 Correct Implementation of the Existing Specification . . . . . 118

6.3.3 The Rao Blackwell Theorem Implementation . . . . . . . . . . 127

6.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128

**7 Evaluation of Recommendation Accuracy 129**

7.1 Evaluation Assumptions . . . . . . . . . . . . . . . . . . . . . . . 130

7.2 Accuracy Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

7.3 Combination Algorithm Tuning Effects on Recommendation Accuracy 132

7.3.1 Robust Singular Value Decomposition Recommender Model . 133

7.3.2 K Nearest Neighbors Recommender Method . . . . . . . . . . 136

7.3.3 Neural Network Recommender Model . . . . . . . . . . . . . 138

7.4 Input/Output Tuning Effects on Recommendation Accuracy . . . . . 141

7.5 Input Data Selection Effects on Recommendation Accuracy . . . . . . 146

7.6 Recommendation Accuracy of the Final Prototype Configuration . . . 147

7.7 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149

**8 Conclusions and Future Work 150**

8.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151

8.2 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 152

8.3 Limitations of the Study . . . . . . . . . . . . . . . . . . . . . . . 155

8.4 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158

**Appendices 163**

**A Input Generation Procedure 163**

**B Matrix Reservation Function 165**

**C Matrix Setup Procedure 167**

**D Load Matrix Procedure 169**

**E Sort Matrix Procedure 171**

**F Truncate Matrix Procedure 173**

**G Save Matrix Procedure 175**

**H Cosine Similarity Procedure 177**

**I Pearson's Correlation Procedure 179**

**J Save Similarities Procedure 181**

**Bibliography 183**

# List of Figures


1.1 Typical Recommender System Input/Output . . . . . . . . . . . . . . 9

1.2 Netflix Dataset Structure . . . . . . . . . . . . . . . . . . . . . . . . 12

2.1 Classification of Recommender Systems [1] . . . . . . . . . . . . . . . 30

3.1 RSVD Matrix Estimation Process . . . . . . . . . . . . . . . . . . . . 37

3.2 Prototype Neural Network Segment . . . . . . . . . . . . . . . . . . . 39

3.3 Netflix Movie Vote Count Distribution . . . . . . . . . . . . . . . . . 43

3.4 Netflix User Vote Count Distribution . . . . . . . . . . . . . . . . . . 43

4.1 Layered Client-Server Architecture . . . . . . . . . . . . . . . . . . . 53

4.2 Layer Interaction During a Recommendation Process . . . . . . . . . 55

4.3 Proposed Component Communication Interfaces . . . . . . . . . . . . 58

4.4 Proposed System Data Flow . . . . . . . . . . . . . . . . . . . . . . . 59

4.5 Recommendation Process with an Embedded Combination Algorithm 62

4.6 Input Generation Algorithm Order of Execution . . . . . . . . . . . . 67

4.7 Input Generation Architecture in Acme ADL . . . . . . . . . . . . . . 72

5.1 Input Generation Process Activities . . . . . . . . . . . . . . . . . . . 75

5.2 Input Matrix Generation – Initial State . . . . . 76
5.3 Input Matrix Generation – Load Matrix . . . . . 77
5.4 Input Matrix Generation – Sorting by Items/Rows . . . . . 77
5.5 Input Matrix Generation – Sorting by Users/Columns . . . . . 78
5.6 Input Matrix Generation – Truncate Matrix . . . . . 78
5.7 A Matrix with Five Vectors and Five Dimensions . . . . . 91
5.8 Truncated Matrices and Their Net Weights . . . . . 92
5.9 Typical Similarities of the First 30 Vectors in an Item-Oriented Matrix 93
5.10 Typical Similarities of the First 30 Vectors in a User-Oriented Matrix 93

6.1 Examples of Inappropriate Minimum Datasets . . . . . 103
6.2 An Appropriate Minimum Dataset . . . . . 104
6.3 Matrix State Changes During the Input Generation Process . . . . . 111
6.4 Matrix Shape Effects on Algorithm Selection . . . . . 112
6.5 Permanent Database Schema . . . . . 113
6.6 Temporary Database Schema . . . . . 113

7.1 RSVD Features and Cycles Tuning . . . . . 135
7.2 RSVD Static Learning Rate Tuning . . . . . 136
7.3 RSVD Learning Rate Reduction Tuning . . . . . 136
7.4 KNN-Item Tuning . . . . . 137
7.5 KNN-User Tuning . . . . . 137
7.6 NN-Item with No Hidden Nodes Tuning . . . . . 140
7.7 NN-User with No Hidden Nodes Tuning . . . . . 140

7.8 NN-Item with 10 Hidden Nodes Tuning . . . . . 140
7.9 NN-User with 10 Hidden Nodes Tuning . . . . . 140
7.10 NN-Item Learning Rate Reduction Tuning . . . . . 142
7.11 NN-User Learning Rate Reduction Tuning . . . . . 142
7.12 Rating Normalization Benefit . . . . . 142
7.13 RSVD Preprocessing Benefit . . . . . 142
7.14 Recommendation Cap Benefit . . . . . 143
7.15 Recommendation Rounding Benefit . . . . . 143
7.16 Recommendation Aggregation Benefit . . . . . 144
7.17 Recommendation Combination Benefit . . . . . 145
7.18 Faster Methods of Selecting Recommender Input . . . . . 146
7.19 More Accurate Methods of Selecting Recommender Input . . . . . 147
7.20 Recommendation Accuracy of the Final Prototype Configuration . . . 148

# Chapter 1

# Introduction

Modern technology advances increased the amount and the rate of information being exchanged online. For instance, broadband Internet access adoption has been consistently rising over the past years [69]. Additionally, previously inaccessible technology has become more affordable, e.g., wireless and mobile Internet. As a result, people spend more time online – as much as 40 additional minutes per day [66]. Such aggressive adoption rates have also influenced the consumption of online content. In fact, faster Internet access can deliver high bandwidth content like entertainment, advertising, images, music, and games. As a result, more and more people use the Internet to access information they need and for many it has become an integral part of their lifestyle.

Browsing the ever-expanding Internet in hopes of finding something interesting can be a never-ending process. The amount of content grows quickly because modern technology makes it easy for anyone to publish online. In fact, only a few reputable organizations had the necessary resources during the early stages of the Internet.

Today, virtually anyone can create a wiki page, post on their blog, or tweet about anything. Millions of people, registered at popular social networking websites, generate terabytes of data in status updates, pictures, videos, and podcasts. With so much information, there is a good chance that one's favorite content exists somewhere on the Internet.

However, the Internet also contains more irrelevant data that makes it progressively harder to find pertinent information. Fortunately, modern computers can quickly scan vast amounts of data for particular keywords and phrases, but unless users know exactly what those keywords are, they will probably not find anything useful. Unfortunately, most people have difficulty describing their preferences to a computer, which is often the reason it takes so long to find relevant information.

Furthermore, computers have a limited ability to understand users' requests. For example, a "bass drum" query makes sense as a musical instrument to another human, but it could mean a "barrel of fish" to a machine. Despite these difficulties, effective search engines do exist, e.g., google.com, bing.com, yahoo.com. These systems successfully search much of the World Wide Web by correctly identifying a user's information need and locating relevant content, despite user errors.

Recommender systems take a different approach to the information overload problem. They do not require a user to know exactly what he/she is looking for or be capable of expressing it in a query. Instead, recommender systems guess user preferences and suggest content that he/she is likely to enjoy. Unlike search engines, recommender systems cannot locate the sought-after content. However, they can recommend serendipitous items that could not be otherwise located [64]. Therefore,

recommender systems do not attempt to replace, but rather complement the information retrieval research [8, 33, 76, 80, 120]. They are especially useful for large content repositories like youtube.com, flickr.com, and wikipedia.com, where users would most appreciate personalized content suggestions.

Recommender system research arose from users' need to process large amounts of information, i.e., the information overload problem. Therefore, the main purpose of a recommender system is to automatically examine large amounts of data and present the user with a more sensible list of recommendations [99]. Naturally, such recommendations are most appreciated for news, where the amount of emerging data may be overwhelming. In fact, some of the first recommender systems prioritized Usenet news messages for individual users [55, 136]. The purpose of modern recommender systems remains the same – they help alleviate the information overload problem and simplify the decision-making process.

## 1.1 Importance of the Study

Recommender systems are becoming increasingly popular online, especially on high-traffic social networking and e-commerce sites. These organizations are literally transforming archives of past consumer behavior into increased revenues and higher retention rates because they can better anticipate their customers' needs [17, 94]. Almost all leading entertainment and e-commerce sites like amazon.com, yahoo.com, and netflix.com employ some kind of a recommender system [147]. Furthermore, many specialized recommenders for web pages, movies, music, and restaurants evolved into

commercial products [62, 64, 124]. Some Internet companies like Net Perceptions are in the business of providing recommendation services [124, 147]. Because there is a commercial demand for good recommendations, we believe this research will find an application.

Even though recommender systems are usually built to improve sales, e.g., suggesting items that a user would not have discovered otherwise, they can also offer non-monetary benefits. For example, many people enjoy browsing recommendations as a guided way to explore the library of available items [63]. Some users enjoy an opportunity to express their opinion and a recommender system provides a forum to do so. Therefore, recommender systems can collect and produce interesting data as well as foster an online community development.

Recommender systems can either make or break the business-customer relationship. Businesses are interested in increasing the amount of products sold, and well-placed recommendations help accomplish that. Consumers want to make smart purchases and try to avoid buying something they might regret later. A trustworthy recommender system can suggest items that consumers will enjoy, thus improving customers' satisfaction and ensuring their loyalty.

User satisfaction is important for commercial applications. In particular, users experience recommendation accuracy and system performance first-hand. For instance, a user will always remember the time when he/she was convinced to buy something as a result of a poor recommendation. Additionally, a user will not bother waiting for a slow recommendation, which could lead to a sale. Accurate and fast suggestions improve user satisfaction, but they are also the most difficult. The following section

list the requirements for a recommender system that can make such suggestions.

## 1.2 Recommendation Accuracy Problem

There is a demand for recommender systems that can consistently produce accurate recommendations, but there are few systems that successfully do so. On the one hand, humans are notoriously unpredictable, and on the other hand, there are technical limitations that prevent detailed dataset analysis. User behavior data is not perfect and there is usually little of it. However, it is often the only source of information available. Therefore, we need a way to infer user behavior patterns from sparse data with limited resources.

The unpredictable nature of human behavior is the reason high recommendation accuracy is so difficult to achieve. In fact, one study showed a natural tendency of its participants to vary their answer when repeatedly asked to evaluate the same set of movies [65]. The authors conclude that each rating contains a random component and that it is impossible to predict true user opinions perfectly. However, opinions are not completely random, and common behavior patterns do exist.

Even though making assumptions and drawing conclusions from a dataset of seemingly random opinions is difficult, good recommendations are possible. In fact, sophisticated probability estimation and behavior-based algorithms can successfully guess users' future preferences based on previous ones. For example, the most successful algorithm can predict a user's rating within a 0.8567 unit on a five-unit rating scale [81]. Furthermore, many systems can recommend over 95% of the items using

only 2% of all possible opinions [16, 142, 171]. Existing algorithms can be accurate, yet they cannot consistently perform on different datasets. Below, we outline desired system requirements that overcome this limitation.

**Interoperability.** Existing recommender systems are usually all-in-one applications that provide one type of recommendation from a single dataset. In many cases, it is difficult to substitute a different recommendation algorithm or make recommendations from a different dataset schema. Such flexibility is often necessary because a combination of different recommendations is usually more accurate than any one of them [23]. Traditionally, this combination has been performed as a post-processing task, so the recommender system would repeatedly search the dataset and construct the same input for a different approach. Consequently, we need an architecture that supports multiple recommender algorithms and eliminates such redundant data processing.

**Performance.** Perfectly adequate algorithms may produce poor recommendations due to physical constraints such as memory and processor limitations. As the number of items and users in the system increase, the performance requirement becomes more critical. Previous recommender system techniques that have been successful for small datasets are not usually adequate, because large datasets have higher storage and processing requirements that may not always be met [3, 63, 142]. Therefore, we need a recommender system that can produce consistently quick recommendations regardless of the dataset size.

**Data Sparsity.** In small datasets, a larger portion of all possible data is available, so

the recommender system has a better understanding of the true user preferences. Even though data sparsity is a common problem, it is especially prominent in large datasets where the amount of available information may not be sufficient to accurately model the user population. Data sparsity is the reason previously successful recommender system techniques fail on large datasets [153]. As a result, we need a new approach that can handle large amounts of data and make more conclusions from little evidence.

**Recommendation Accuracy.** The usefulness of the entire recommender system relies on the accuracy of suggestions it makes. However, humans are difficult to predict because, unlike machines, they exercise free will. No system can perfectly predict future user behavior, no matter how much historical data is available. However, it can be wrong less often and create an illusion of predictable future. Even a small accuracy improvement in a system with thousands of users would result in a tremendous revenue increase. Therefore, we need a recommender system that can meet accuracy expectations of modern applications.

Performance and accuracy are opposing qualities, as good recommendations generally take a long time to generate. However, prediction accuracy is a more important problem to address because users can tolerate waiting for consistently good results instead of receiving bad recommendations instantly [142]. We focus this research on improving recommendation accuracy, while addressing data sparsity and technical limitations as causes of this problem.

## 1.3 Research Hypothesis

Recent studies show that it is increasingly difficult to make substantial progress in prediction accuracy. For instance, a 1% improvement on the Netflix Prize challenge took nearly two years [16]. We believe it is because most effort has been directed toward optimizing algorithms instead of improving their input data. One of the most common ways to do so is to supplement the dataset [63]. However, users are usually hesitant to provide additional ratings, third party data might not be available, and implicit rating analysis is unreliable [12, 102]. Accordingly, we do not add more data, but remove data we know is irrelevant. The main contribution of our work is an algorithm that identifies only the most relevant data for recommender input. Even though it is difficult to make accurate assumptions from little evidence, we believe that better input selection can lead to more accurate recommendations, regardless of the algorithm.

## 1.4 Conceptual Model of a Recommender System

The goal of a recommender system is to learn user preferences from the past and apply this knowledge to predict the future. Figure 1.1 demonstrates the input and output of a typical recommender system. It trains on a set of known ratings and produces predictions for a set of unknown ones. At this level, the exact calculations inside a recommender system are irrelevant, as long as it produces suggestions in the desired format.

More formally, the recommender system may be described as a set of $m$ users

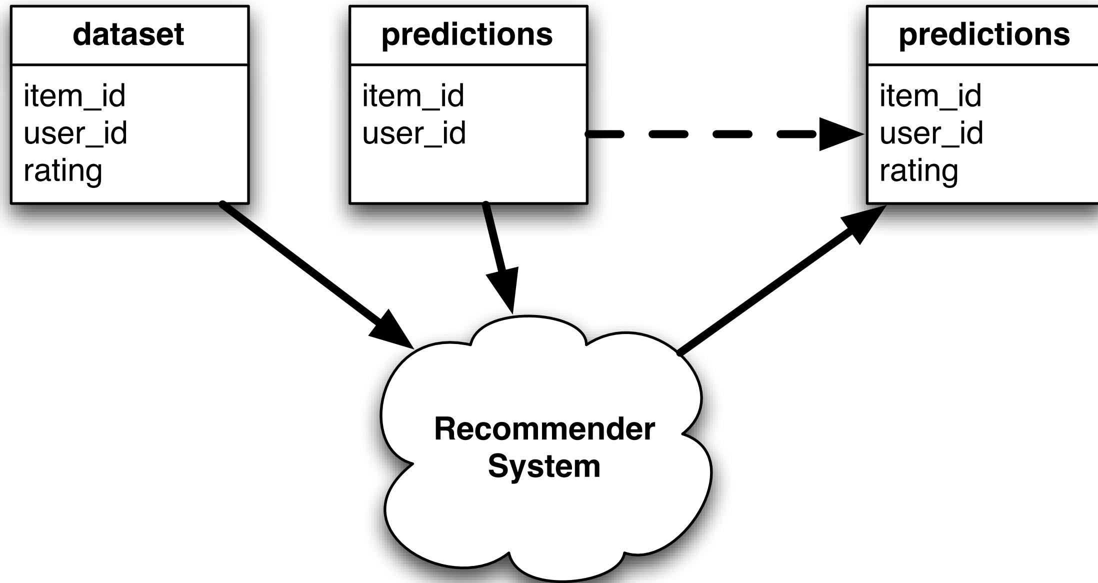


Figure 1.1: Typical Recommender System Input/Output

$U = \{u_1, u_2, ..., u_m\}$ and a set of $n$ items $I = \{i_1, i_2, ..., i_n\}$. Each user $u$ has an associated set of items $I_u \subseteq I$, which he/she has rated. Each rating $r$ is assumed to be on a discrete numerical scale, even though continuous rating scales are also common [112]. The user and item for which the prediction is to be made are called *active user* and *active item* [29, 142]. A recommender system guesses the opinion of user $u$ on item $i$, $P_{u,i}$. Usually, production systems predict opinions that have not been previously recorded, $P_{u,i} \mid i \notin I_u$. Development systems guess a set of known ratings, so that the error of each prediction may be computed, $P_{u,i} - r_{u,i}$.

## 1.5 Recommendation Accuracy Metrics

The most appropriate way to evaluate the quality of a recommender system is to survey its users. Their confidence and satisfaction with recommendations are two

qualitative ways of evaluating a recommender system. However, the only way to acquire such high quality feedback is through live human evaluations, which are difficult to arrange and provide a limited amount of test cases.

An alternative way to evaluate a recommender system is to quantitatively measure its accuracy. One popular measure is the Mean Absolute Error (MAE), which is the average absolute deviation between a predicted user opinion and the actual one. It is a simple measure that gives all errors the same importance, regardless of their size. Another popular measure is the Root Mean Squared Error (RMSE), which is the square root of the average of squared deviations [29, 142]. It is a slightly more complex measure that is more sensitive to large errors. In fact, RMSE values are usually the same or slightly greater than MAE values because the squaring process gives larger errors more importance. The formulas for MAE and RMSE are as follows:

$$MAE = \frac{\sum\limits_{u \in U, i \in I} |P_{u,i} - r_{u,i}|}{|P|}$$

$$RMSE = \sqrt{\frac{\sum\limits_{u \in U, i \in I} (P_{u,i} - r_{u,i})^2}{|P|}}$$

Usually, it is best to choose a metric that uses the same units as data, i.e., it represents the size of a typical error. Both MAE and RMSE satisfy this criteria. Additionally, they are both negatively oriented, so lower values are considered better. However, neither measure has an absolute value that is considered best [64, 142, 146]. Instead, recommender systems are ranked according to their typical error size within a dataset.

The creators of our dataset deem large errors to be particularly undesirable, i.e., the cost of an error is greater than its size. Therefore, the RMSE measure is most appropriate for our dataset. Furthermore, because it is impossible to compare accuracy on different rating scales, recommendations on the same dataset use the same metric. Our dataset has a list of 130,000 RMSE scores available [16]. Therefore, we use the RMSE measure to evaluate our prototype.

## 1.6 Source of the Experimental Data

We evaluate the fitness of our recommender prototype on a recent and widely published dataset provided by Netflix. In fact, Task 1 of the leading Data Mining and Knowledge Discovery competition in the World (KDD CUP 2007), is based on this dataset [15]. One of the most obvious reasons for such popularity is the dataset size. It contains over 100,000,000 actual movie ratings on a discreet scale from one to five. It represents opinions of over 480,000 users and almost 18,000 movies [15]. It is more than 30 times larger than any other available dataset. However, it represents only 1.1% of all possible ratings [15], so approaches that rely on a higher data density may not apply. We choose to use this challenging dataset because it is large, sparse, and it has a number of published RMSE scores.

Before the Netflix dataset was available, researchers used a number of smaller and denser datasets. The three most published datasets are EachMovie, MovieLens, and Jester [63]. EachMovie is the most common of the three. It contains over 2,800,000 movie ratings on a discreet scale from zero to five from almost 73,000 users on 1,600

movies [9, 31, 63]. Only 2.4% of all possible ratings are captured in this dataset. The MovieLens dataset extracts usually contain 100,000 ratings on a discreet scale from one to five from about 900 users and 1,500 movies [3, 27, 40, 142]. Only 6.3% of all possible ratings are captured in this dataset. A more recent dataset comes from Jester joke recommendation website. Some of the most popular versions of this dataset contain almost 900,000 ratings on a continuous scale from -10 to +10 from 17,000 users on 100 jokes [87, 112]. This dataset contains over 50% of all possible ratings. No previously listed dataset provides the size or the sparsity level of the Netflix dataset.

The structure of our dataset is fairly standard. Figure 1.2 shows how the information about every movie is available in a single file. The ratings are grouped by movie and stored in separate files. The dataset contains a list of 2,800,000 withheld ratings (qualifying/test set). Netflix evaluates estimates of these opinions. The dataset also contains 1,400,000 known ratings (probe/quiz set) intended for local evaluation. We use the quiz set for our case study.

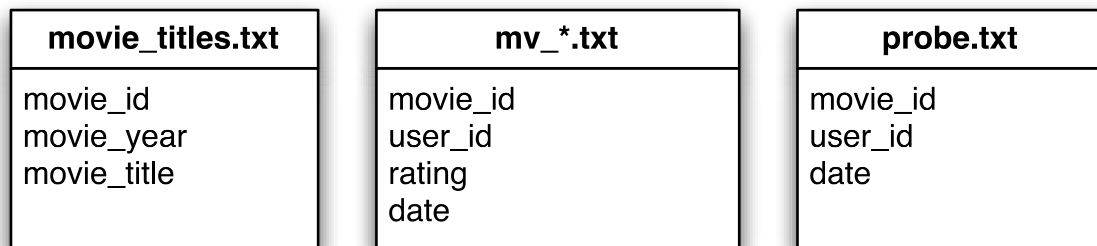


Figure 1.2: Netflix Dataset Structure

## 1.7 Contributions and Organization

Data sparsity is often cited as the primary reason for poor recommendations, yet we believe that accurate recommendations can be made with little data. The purpose of this research is to find a way for a recommender system to train on a sparse dataset and predict future user opinions with maximum accuracy. Our work is documented in the following chapters that contain theoretical and technical contributions for current and future recommender system research.

In Chapter 1, we establish the motivation, importance, and background of our work. We then outline the evolution of recommendation techniques and present modern ways of making personalized suggestions in Chapter 2. We discuss different ways such recommendations may be done and identify common obstacles within each approach. Content-based recommendations suffer from the finite nature of machines, whereas collaborative filtering approaches monopolize on human ability to comprehend abstract concepts. For this reason, we conclude that collaborative filtering approach is more applicable for further investigation.

Chapters 3 and 4 focus on collaborative filtering recommendations. Chapter 3 provides a survey of popular algorithms, describes how they generate recommendations, and compares their reported accuracy. We later implement three popular combination algorithms described here. Chapter 4 formulates the requirements of a scalable system that can efficiently produce multiple types of recommendations. We also propose an architecture that satisfies these requirements. This architecture is necessary for our experiments, because it allows recommender input to be computed asynchronously to the recommendation process. As a result, we can perform quick

experiments with different algorithms without regenerating the same input.

The main hypothesis of this research is that a small number of relevant ratings is sufficient to make an accurate recommendation. We believe that such ratings may be chosen with local similarity, instead of a more traditional global similarity. Chapter 5 compares two ways of computing vector similarity. We describe the standard process of generating recommender input and make some observations regarding the desired qualities of such input. Finally, we present a statistical justification for input resorting as a way to improve recommender input quality.

Our novel input generation algorithm implements the input resorting approach. The input quality is the essence of our research because it has the most influence on recommendation accuracy. To encourage future development around the input generation component, we formalize its design in Chapter 6. We use the Z notation to identify the restrictions that must be met at each stage of the input generation process. Such formal model of the component behavior serves as a blueprint for implementation. We also use the model to confirm that the implementation corresponds to earlier specification. We verify its compatibility, consistency, correctness, and completeness.

To determine the effectiveness of our hypothesis, we perform an empirical evaluation of three popular combination algorithms in Chapter 7. We examine their accuracy over a wide range of configurations. Some of these algorithms are more effective because of the rating patterns in a particular dataset. We analyze the Netflix dataset and outline such patterns. Additionally, we evaluate the claim that a committee of recommenders produce better results than any one of them. We also

test the benefit of normalizing the input and output of a recommender. Finally, we examine the benefit of inout quality on recommendation accuracy.

We conclude our research with the discussion of our work and suggestions for future research in Chapter 8. Our results show that tuning recommender algorithms has little effect on recommendation accuracy. However, all methods produce better predictions when supplied with more relevant input data. Our recursive input generation approach sorts the input twice to decide which ratings are relevant. This method is the reason our prototype can achieve better recommendation accuracy. The results support our hypothesis that good suggestions may come from little evidence.

## 1.8 Conclusion

Recommender systems are software applications that help address the problem of information overload. They go through large amounts of content and recommend only the most interesting items. Recommender systems are particularly useful for commercial applications, where customers enjoy the personal shopping assistance and stores increase sales. This research is important to both parties, as people who produce and receive recommendations prefer them to be accurate. Many existing algorithms can successfully predict opinions with reasonable accuracy. However, as the amount of content grows, we need more sophisticated systems that consistently match unknown user opinions within a small margin of error. The next chapter describes the different techniques capable of such recommendations.

# Chapter 2

# An Overview of Recommender Techniques

The problem of information overload is not new, and there are many techniques that address this subject. In fact, recommender system research developed alongside other related computer science disciplines. For example, Herlocker shows how personalized suggestions relate to information retrieval, information filtering, and machine learning work [64]. Ricci and Werthner further elaborate on the historical overlap of recommender system research with artificial intelligence and human-computer interaction [138]. In Figure 2.1 we show how Adomavicius and Tuzhilin [1] classify many known recommenders in terms of well-established computer science techniques. We summarize the existing research on recommender systems in this chapter. In particular, we discuss the progression of recommender systems from original implementations to modern state-of-the-art techniques.

Early recommender systems were basic and lacked personalization. Such systems

aggregated community opinions and produced global recommendations [64, 98]. For example, the *New York Times* Best-Seller List suggests popular books ranked by their sales volume. These items are popular and many people enjoy them, however popular content does not necessarily appeal to all users. To alleviate this problem, some recommendations are organized in categories, e.g., fiction, nonfiction, and children's books. Such categorized suggestions also have limited personalization because they assume that complex user preferences may be characterized by a single book genre.

The first personalized recommender system was called Tapestry, an e-mail recommender system [55]. It allowed users to leave comments regarding the perceived value of messages and filter topic discussions based on specified criteria. Unfortunately, this system was not easy to use because it relied on a proprietary query language. Furthermore, Tapestry could not automatically identify users with similar interests. Instead, each user had to manually set up a list of trusted peers [55, 124, 142]. The Tapestry project was meant to be used in a small community, where every user personally knew everyone else. Therefore, it could not be scaled to larger organizations with thousands of users.

The first automated recommender systems, like GroupLens [136] and Ringo [148], automatically identified potentially useful sources of recommendations. In fact, the GroupLens project was the first to introduce similarity measures as a way to establish trust and reputation among users. In this case, trust represents the perceived competence of a user providing a recommendation for a particular item [168]. However, such concept is complex and difficult to model. As a result, modern recommender systems often employ simplified versions of such relationships, e.g., rating overlap [116, 118].

Despite these sacrifices, similarity measures made automatic recommendations possible. The following three sections outline common personalized recommendation approaches.

## 2.1 Content-Based Recommendations

Content-based systems recommend items that are similar to items that the user has previously liked. The assumption is that once a user has expressed interest in an item, he/she is likely to enjoy a different item with similar content. For example, SiteHelper actively communicates with the user, asking for document keywords and their importance [113, 167]. This system helps the user formulate a search query by suggesting new keywords. Letizia is a similar recommender system, except it locates similar documents in the background, while the user reads [91]. It assumes that since the user bothers to inspect a document, it must be interesting. These systems focus on two different types of feedback, but provide purely content-based recommendations and rely on their ability to recognize and locate keywords. Therefore, the ability of content-based recommenders to categorize content is essential.

### 2.1.1 The Limited Content Analysis Problem

Content-based recommender systems require detailed information regarding items and a way to compare them. For instance, restaurant recommender systems need to understand the difference between various cuisines [1, 28, 70]. Learning that kind of information is difficult, especially in abstract domains. Alternatively, content may

be described in a set of features that are either manually assigned or automatically extracted from machine-readable content [50, 58, 161]. Unfortunately, even the most sophisticated data analysis techniques cannot produce a complete description of certain items [82, 84, 126]. For example, a web page may be parsed for keywords and links, but purely aesthetic qualities such as image content, embedded music/video, loading time, and page layout are ignored [5]. As a result, an inappropriate or illegible web page may be recommended to a user just because its keywords match his/her preferences.

The limited nature of item descriptions is the root of poor content-based recommendations. Such systems may not distinguish between two different items that have the same set of features, e.g., two articles on the same topic, with the same set of keywords, but opposing viewpoints [5, 148]. As a rule, content-based systems cannot capture abstract properties such as publisher's reputation, quality of references, writing style, and author's point of view.

To overcome this, a system may introduce more features to better describe each item, but doing so does not address the root of the problem. Computers cannot accurately characterize abstract attributes, regardless of how many features are available. Some systems employ human experts to fill in abstract features [1, 148]. However, manual feature assignment by a single person is likely to be biased and a committee evaluation would be too slow. Therefore, this approach is not applicable for large systems with thousands of items.

### 2.1.2 The Overspecialization Problem

Content-based recommender systems can recommend known favorites, but cannot make serendipitous recommendations. This phenomenon is usually referred to as the overspecialization problem [1, 64, 148]. It occurs when recommendations form a homogeneous set of interchangeable suggestions as opposed to good recommendations that are diverse and cover a large number of potentially interesting topics.

The source of the overspecialization problem may be the dynamic nature of user preferences. A recommendation process that assumes static preferences will fail to accurately model user behavior. To accommodate this limitation, some systems employ a genetic algorithm, i.e., an artificial ecosystem of competing and cooperating agents that represent users' continuously evolving interests [108, 149]. This kind of "survival of the fittest" approach reduces the overspecialization effect by creating a dynamic model. However, maintaining a complex model like this for thousands of users may not be feasible.

Other solutions emphasize data age as the cause of recommender overspecialization. For instance, some believe that recommendations based on the most recent feedback form a more accurate representation of users' changing preferences [24, 45, 46]. Thus, introducing a positive bias toward more recent user feedback or establishing a time-window that limits relevant votes should improve accuracy. One solution that implements the latter approach is Daily-Learner, a news classification system that monitors users' short-term and long-term interests [24]. It limits the amount of recommendations that are too similar to existing short-term preferences, while suggesting items that are similar to long-term favorites [1]. As a result, its recommendations

change continuously, while still appealing to users' core tastes.

Content itself may also cause overspecialization. Zhang, Callan, and Minka argue that documents created at or around the same time are more likely to contain redundant information [173]. Therefore, suggested items should be similar to previous recommendations, yet offer new information. However, this work is limited to news recommendations, where same events are likely to be covered in various sources around the same time. The proposed measures of novelty and redundancy do not apply to art, where multiple highly acclaimed works may be released during the same period. Therefore, each content-based system must deal with limited content analysis and overspecialization problems within their domain.

### 2.1.3 The New User Problem

Content-based systems can recommend items with interesting content, even before anyone rates them. However, they cannot recommend content to new users, who have few or no preferences. Previous rating history helps justify future recommendations, so a user with no history has no established preferences and no recommendations. Furthermore, the overspecialization problem is more substantial for new users with very few favorite items because the system assumes that previously rated content is the only type of content they enjoy [1,134]. The new user problem is also common in collaborative filtering systems. We discuss it in more detail in the following section.

## 2.2 Collaborative Filtering Recommendations

Collaborative filtering systems recommend items that other users enjoyed, essentially automating the "word-of-mouth" suggestions [64, 136]. The assumption is that one user's favorite items may be inferred by observing other users with similar interests. As the name implies, an algorithm derives personalized recommendations from filtering all available items through preferences of similar users [5]. Unlike the content-based approach, which relies on item similarity, this approach computes similarity between users, i.e., shared opinions.

However, user opinions are subjective and have little to do with content similarity. In fact, a pure collaborative filtering system has no knowledge of item content, which makes it ideal for abstract domains, e.g., paintings, music, and poetry [39, 86]. The lack of content analysis also allows collaborative filtering recommenders to make serendipitous suggestions. If another user enjoyed a particular item, it may be recommended to you, despite the fact that you have never expressed interest in such content.

### 2.2.1 The Sparsity Problem

Recommendations made from a sparse dataset lack clarity and certainty because there is little evidence to justify them, i.e., it is difficult to find similar users in sparse data [79, 122]. Modern datasets contain thousands of items, so it is unlikely that any user will rate every item. However, there exists a subset of users, called the critical mass, that contributes a sufficient amount of ratings [1,5]. Before a system reaches the

critical mass, it is difficult to produce accurate recommendations [71,142]. Therefore, collaborative filtering recommendations may not apply to extremely sparse datasets.

The most popular way to reduce data sparsity is to add default ratings to the dataset. Some of the easiest default ratings to compute include mean item rating, mean user rating, majority item rating, and majority user rating [29]. However, aggregate defaults are usually poor approximations of the actual user opinions. They generalize unique preferences, so they are usually neutral or negatively skewed to ensure a more conservative prediction [26,63].

Often, default ratings come from external sources. For example, the MovieLens project populated missing values with existing ratings from a different movie dataset [146]. Likewise, Basu, Hirsh, and Cohen used the Internet Movie Database website to supplement their dataset [10]. Using external sources of default ratings is a simple and effective way to reduce data sparsity, but such resources may not always be available.

Implicit user feedback is always available, which makes it a popular way to supplement a sparse dataset. For example, the GroupLens research group used the time spent reading a message as an indicator of preference [103]. Likewise, the PHOAKS system used links in Usenet messages [157] and the Siteseer system analyzed browser history [139] to identify user interest. However, just because a user has expressed interest in an item does not mean he/she necessarily liked it. For example, even though bookmarking a page would generally be considered as an indication of interest, it could also mean that the user does not have time to read a potentially irrelevant page [167]. Therefore, implicit feedback is an abundant, but not reliable source of

default data.

The Singular Value Decomposition (SVD) algorithm can estimate missing data in a sparse dataset. It is a well-known dimensionality reduction technique that can estimate a dataset modeled as a matrix, where each cell contains a numeric value representing a user's vote on a particular item [1, 23, 133, 145]. The resulting estimate is a complete matrix that may supplement a sparse dataset or produce recommendations [16, 112, 140, 145]. Unfortunately, the original SVD approach is meant to approximate a complete matrix [13]. However, there are more robust SVD approaches [101, 125] that can accurately estimate a matrix with some of its ratings missing.

### 2.2.2 The New User Problem

The new user problem occurs when a recommender system cannot make a recommendation for a user with little or no preference history. The easiest solution is to recommend universally liked items. Such suggestions can be accurate, but only in small subdomains with a relatively static ranking of best items [97]. Universally liked items may not appeal to everyone in a larger domain, where individual user preferences are less likely to be aligned. However, many domains have a myriad of subjective categories that often overlap, e.g., music. Therefore, making suggestions to new users in such domains would be difficult.

Collaborative filtering systems may fail to recognize the similarity of two new users who agree on the same kind of content because they have not rated the same items [118]. Huang, Chen, and Zeng propose a way to locate similar users through

transitive properties of rating histories [1]. For example, if Anne and Bob have some ratings in common, while Bob and Charlie also share some ratings, a conventional approach ignores a possible relationship between Anne and Charlie. The authors propose a way that relates Anne to Charlie through Bob, which is a more natural way to compare users. This approach does not change the dataset, but improves the way data is selected.

Alternatively, one can convert a new user into an established user through an expedited orientation. For example, Rashid et al. force all new users to answer a few key questions [134]. This way the recommender system has the initial knowledge on which to base its suggestions. The choice of questions determines the effectiveness of this approach. The goal is to minimize the burden on the user, by asking fewer questions, and maximize the system's understanding of the user's preferences, by covering a broad range of interests [77, 124]. After comparing different ways to select questions, the authors recommend asking new users to rate popular items with a wide rating variance.

However, new users may be hesitant to fill out long questionnaires. Therefore, some systems learn faster by noting which questions the users skip [134, 167]. Furthermore, Yu et al. [171] implement an algorithm that considers user like-mindedness to ask questions a person is most likely to answer. As a result, new users are not forced to answer as many questions and the system has enough information to produce initial suggestions.

New user recommendations can also be inferred from a cluster of users with similar tastes [57,115]. For example, the GroupLens project clustered users together based on

their preferences [79]. As a result, each cluster appeared as a user with a rich rating history. Such clusters provide good default ratings, but require a way to determine user membership. In order to group users with no expressed preferences, some systems consider additional properties, e.g., age, gender, and education [1,123]. This approach is an effective way to reduce sparsity and improve performance [44, 133, 144, 162]. However, clustering similar data prevents personalized recommendations.

### 2.2.3 The New Item Problem

When new items are added to the system, few users have had a chance to rate them. Since the collaborative filtering approach makes recommendations based on what other users liked, new items do not have enough supporters to be considered [1, 18]. This is particularly problematic for recommender systems with a continuous flow of new arrivals, as potentially great new items are likely to be disregarded in favor of older ones.

One solution is to employ several rating programs that populate the dataset with default values. Such programs, called filterbots, represent legitimate users of a collaborative filtering system [56, 134]. In fact, sparse datasets enhanced with a set of simple filterbots, e.g., all comedies receive a four star rating, provide better results than users alone. They accelerate the new item settling process by generating ratings users are likely to provide. As a rule, individual filterbots are usually poor predictors, but a collection of them could reduce dataset sparsity without introducing much noise.

Items may also be clustered according to their content. For instance, GroupLens

clustered related news articles [79]. Once clusters are established, a new item receives a set of default ratings approximated from items in the same cluster [29, 40, 162]. However, content classification relies on domain-specific knowledge, which is rarely available [134, 142]. Alternatively, items may be clustered according to their ratings [119], but such clusters may not have clear-cut boundaries. Therefore, clustering new items may get them noticed faster, but it does not guarantee that they will be recommended properly

Clustering items by content is similar to the hybrid approach that combines the content-based and collaborative filtering methods. For example, one algorithm uses sentiment analysis to estimate a numeric opinion from an unstructured, natural language review [89]. The result is a collaborative filtering recommendation based on the content-based input.

## 2.3 Hybrid Recommendations

Hybrid recommendations combine content-based and collaborative filtering methods. In fact, both pure approaches may be considered as special cases of the hybrid method [5, 150]. If the content-based component cannot find any items with similar content, the recommendation becomes a purely collaborative filtering problem. If the collaborative filtering component cannot find any users with similar preferences, the recommendation becomes a purely content-based problem. Many hybrid recommenders are more accurate than pure content-based and pure collaborative filtering approaches, especially in the context of new user and new item prob-

lems [5, 39, 123, 152, 162]. Such systems are more robust and suffer from fewer drawbacks, but they are considerably more complex and difficult to maintain.

Keeping content-based and collaborative filtering suggestions separate simplifies the system. This way each component can specialize in a particular kind of recommendations. For example, ProfBuilder is a website recommender that provides two separate lists of recommendations [134]. Content-based list contains other websites on the current topic, even if they have not been visited before. Collaborative filtering list contains websites other people visited, even if they are on different topics [167]. This way the user always has some recommendations, even if one list is empty.

However, user experience may improve if he/she does not have to choose between two sets of recommendations. One way to automate this choice is to choose the best available recommendation according to the biggest confidence measure or biggest congruency with existing user ratings [1]. Unfortunately, these metrics are inconsistent and cannot guarantee the choice correctness.

Instead of choosing the best one, multiple recommendations may be combined together for a more consistent result. For example, Claypool et al. propose a hybrid recommender system that combines recommendations as a weighted average of collaborative filtering and content-based suggestions [39]. Pazzani also studied the different ways to combine five recommendation methods [123]. His experiments showed that combined results were in fact better than any one method. Computing different recommendations independently and combining them as the final step has definite advantages. A practitioner may choose the leading implementation in each category and aggregate the results without the burden of developing a comprehensive

recommendation algorithm.

Alternatively, the collaborative filtering and content-based recommendations may be fused together into a unified model that considers them both. Popescul et al. perform experiments with such a recommender on extremely sparse datasets and show it to be more effective than pure collaborative filtering and pure content-based recommenders [129]. However, developing and improving such a system is tedious because it is large, complex, and tightly coupled.

## 2.4 Conclusion

This chapter provides an overview of the main recommender techniques and problems that are inherent to each approach. We believe that the collaborative filtering method is most appropriate for our research. It can recommend any type of content and has milder drawbacks. In fact, all disadvantages are related to data sparsity. In many cases, asking users a few additional questions may solve the issue. However, we focus on cases where the dataset may not be supplemented by users or content analysis. This is one of the most common and most difficult problems in recommender systems research. The next chapter outlines the popular solutions to this problem.

| Recommendation Approach | Recommendation Technique | |
|---|---|---|
| | Heuristic-based | Model-based |
| Content-based | Commonly used techniques:<br>• TF-IDF (information retrieval)<br>• Clustering<br>Representative research examples:<br>• Lang 1995<br>• Balabanovic & Shoham 1997<br>• Pazzani & Billsus 1997 | Commonly used techniques:<br>• Bayesian classifiers<br>• Clustering<br>• Decision trees<br>• Artificial neural networks<br>Representative research examples:<br>• Pazzani & Billsus 1997<br>• Mooney et al. 1998<br>• Mooney & Roy 1999<br>• Billsus & Pazzani 1999, 2000<br>• Zhang et al. 2002 |
| Collaborative | Commonly used techniques:<br>• Nearest neighbor (cosine, correlation)<br>• Clustering<br>• Graph theory<br>Representative research examples:<br>• Resnick et al. 1994<br>• Hill et al. 1995<br>• Shardanand & Maes 1995<br>• Breese et al. 1998<br>• Nakamura & Abe 1998<br>• Aggarwal et al. 1999<br>• Delgado & Ishii 1999<br>• Pennock & Horwitz 1999<br>• Sarwar et al. 2001 | Commonly used techniques:<br>• Bayesian networks<br>• Clustering<br>• Artificial neural networks<br>• Linear regression<br>• Probablistic models<br>Representative research examples:<br>• Billsus & Pazzani 1998<br>• Breese et al. 1998<br>• Ungar & Foster 1998<br>• Chien & George 1999<br>• Getoor & Sahami 1999<br>• Pennock & Horwitz 1999<br>• Goldberg et al. 2001<br>• Kumar et al. 2001<br>• Pavlov & Pennock 2002<br>• Shani et al. 2002<br>• Yu et al. 2002, 2004<br>• Hofmann 2003, 2004<br>• Marlin 2003<br>• Si & Jin 2003 |
| Hybrid | Combining content-based and collaborative components using:<br>• Linear combination of predicted ratings<br>• Various voting schemes<br>• Incorporating one component as a part of the heuristic for the other<br>Representative research examples:<br>• Balabanovic & Shoham 1997<br>• Claypool et al. 1999<br>• Good et al. 1999<br>• Pazzani 1999<br>• Billsus & Pazzani 2000<br>• Tran & Cohen 2000<br>• Melville et al. 2002 | Combining content-based and collaborative components by:<br>• Incorporating one component as a part of the model for the other<br>• Building one unifying model<br>Representative research examples:<br>• Basu et al. 1998<br>• Condliff et al. 1999<br>• Soboroff & Nicholas 1999<br>• Ansari et al. 2000<br>• Popescul et al. 2001<br>• Schein et al. 2002 |

Figure 2.1: Classification of Recommender Systems [1]

# Chapter 3

# Collaborative Filtering Algorithms

This chapter presents a literature review on collaborative filtering recommenders. These systems suggest items that similar users enjoyed because people who agreed in the past are likely to agree in the future. The collaborative filtering approach can recommend new and interesting items that content-based systems fail to recognize due to their overspecialization tendency. In fact, item content is completely irrelevant, because collaborative filtering recommendations are based exclusively on user opinions. We focus on this approach because it applies to a wide variety of domains. The following sections describe two types of collaborative filtering recommenders and explain how they produce personalized suggestions.

## 3.1 Model-Based Algorithms

Model-based systems create an offline model to represent the dataset. This provides a considerable online performance improvement, because the dataset is not referenced

for making recommendations. Instead, a model can quickly generate predictions according to the conjectures encoded within it [67, 83, 142]. Furthermore, a model may be used to understand and imitate user behavior through highlighting preference correlations and rating patterns [124]. However, a model is a generalized approximation of original data, with assumptions that may not be true for the entire dataset. In fact, model accuracy is proportional to the amount of data missing [26, 74].

A model can concisely represent a large dataset, yet model creation is a resource-intensive task. It is usually performed offline, which prevents new data from influencing recommendations until after the model is rebuilt [29, 99]. These drawbacks affect recommendation accuracy, so we need a simple model that can capture many detailed conjectures. Below is an overview of how some collaborative filtering systems model their datasets. We discuss four different approaches but focus on the latter two, because research shows them to be more accurate.

### 3.1.1 Association Rules Model

The association rules approach applies rule discovery algorithms to detect rating patterns and associations among items [10, 90, 93, 142]. The first recommender system to use association rules was Lens, an email recommender system [96]. There, each user specified a set of conditions an email had to meet in order to be read. These manual rules made up the social filtering component that was responsible for recommending only the most important emails.

Modern association rule models discover such rules automatically. For instance, Mobasher et al. developed a web page recommending algorithm that identified asso-

ciation rules by mining users' browser histories [107]. However, this algorithm only used high confidence associations, so it did not produce many of them. In fact, many association rule models generate too many irrelevant or too few solid rules because they use a static support threshold [141, 143]. Lin, Alvarez, and Ruiz propose a method that produces more rules by adjusting the required support for individual users [92]. This approach guarantees association rule discovery at the cost of variable certainty.

Association rules provide a clear and concise explanation for each recommendation. This is an attractive characteristic for many applications, but this model fails to acknowledge the user's individuality and lacks personalization that other models offer. Despite this, versions of association rules model can make accurate recommendations, e.g., clustered association rules model [43, 44]. Therefore, association rules model is not effective on its own, but it can be combined with other approaches to improve accuracy.

### 3.1.2 Probabilistic Models

Probabilistic models use probability distributions to encode and make conclusions about the dataset. For instance, Bayesian networks represent the dataset as a set of interdependent nodes, where the order in which users rate items establishes node dependencies [26, 44]. Alternatively, nodes can represent users and edges between nodes – their predictability [2,174]. Each node has several states for ratings and their probabilities. Therefore, following a path that best describes the active user will lead to a recommendation. This model ca make quick recommendations and it does not

require a long learning phase.

Because of their simple structure, Bayesian network models can produce fast results. However, as the amount of items and users increases, the model grows exponentially. In fact, if new users and items are added frequently, the model may be too large to rebuild every time. One solution is a dependency network with bidirectional edges [44, 59]. It is not as accurate as the Bayesian approach, but it learns more efficiently and requires less space. Alternatively, a model does not have to store every possible path, but instead record only the best ones [26,142]. This reduces the storage requirement, but does not reduce the amount of computations.

Bayesian clustering addresses this problem by grouping similar nodes together. This approach assumes that all users may be distinguished across a small number of predefined tastes. Since the tastes are independent, a recommendation may be inferred from a set of participating clusters that best describe the active user [10, 74, 162]. The probability of a user belonging to a particular cluster as well as the individual cluster preference distribution is usually observed from empirical data [26, 44,142]. This approach is compact, fast, and it may be updated continuously, without reducing performance. However, it does not produce personalized conclusions because users within a cluster are indistinguishable.

Similarly to Bayesian clustering, the personality diagnosis model also classifies users, but it treats each user as a separate model. Each rating is assumed to be drawn from a normal distribution centered around the item's true rating [74,124]. The recommendation is then a product of independent rating probabilities from users who already rated the item. This model is more accurate than the Bayesian network and

clustering approaches. However, since it relies on normal distributions, the accuracy may suffer from the new item problem, i.e., less than 30 ratings per item.

The aspect model also assumes that every rating is a combination of multiple probability distributions. Unlike previous attempts that focused exclusively on users or items, it represents each rating with three preference factors: the base probability of a rating occurring anywhere in the dataset and probabilities that this user and item will have such a rating [68]. However, users with similar opinions do not necessarily rate items the same way. Jin, Si, and Zhai extend this model by introducing two more hidden variables: the probability that a user has the same preferences and the probability that these preferences are expressed the same way [74]. The experiments on the EachMovie dataset show that this model performs better than Bayesian and the original aspect model.

### 3.1.3 Singular Value Decomposition Model

Singular Value Decomposition (SVD) model approximates a matrix of ratings (*users* × *items*) as a product of two smaller matrices (*users* × *features* and *items* × *features*). One of the most popular versions of this approach, Robust SVD (RSVD) may be trained incrementally on a sparse dataset. Because this approach examines all signals, including the weak ones, it provides a more complete model of the dataset [12]. Furthermore, it requires little storage and does not involve complex operations, which makes the RSVD model an effective, fast, and scalable solution for large datasets. In fact, one of the biggest improvements in the Netflix Prize challenge has been due to this algorithm [15, 140].

The goal of this approach is to produce an estimate that most closely resembles existing data. It repeatedly adjusts values in composite matrices until the individual approximation error is minimal. This error is the difference between the actual rating and the product of user and item features:

$$E = r_{i,j} - x_{i,k} * y_{j,k}$$

The best estimate is reached by following the gradient of an error function (MSE) across a set of known ratings, $R$. Because the amount of known ratings is constant, we can simplify the error function and determine its gradient. The formulas for the original MSE, the simplified error function, it's gradient, and two partial derivatives are as follows:

$$MSE = \frac{\sum E^2}{|R|}$$

$$F(x_{i,k}, y_{j,k}) = E^2$$

$$\nabla F(x_{i,k}, y_{j,k}) = (2E * dE/dx_{i,k}, 2E * dE/dy_{j,k})$$

$$dE/dx_{i,k} = -y_{j,k}$$

$$dE/dy_{j,k} = -x_{i,k}$$

Note that partial derivatives of the error function show the relationship between item and user features. In other words, an approach that focuses on only one kind of vector will ignore an important part of this dynamic relationship. Finally, the model updates composite matrices in the direction opposite of the gradient, thus adjusting their product to reduce the approximation error. In order to speed up this process, adjustments are multiplied by a learning speed, $\gamma$:

$$x_{i,k} = x_{i,k} + \gamma(2E * y_{j,k})$$

$$y_{j,k} = y_{j,k} + \gamma(2E * x_{i,k})$$

Figure 3.1 shows the RSVD approximation of a 2 × 2 matrix after 10 and 100 iterations. Our goal is to estimate it as a product of two smaller 2 × 1 matrices with a 0.1 learning speed. Since the rating scale is discrete, we can round the model estimate to achieve a perfect matrix approximation. We implement this model as one of our combination algorithms.

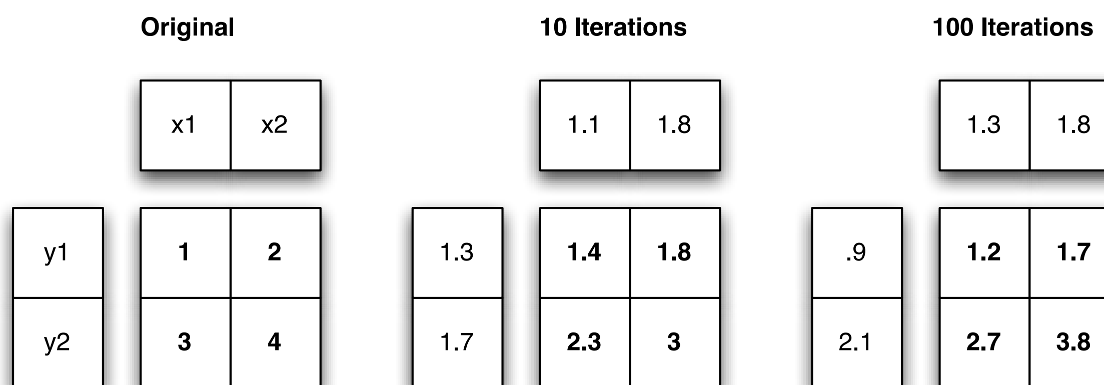


Figure 3.1: RSVD Matrix Estimation Process

### 3.1.4 Neural Network Model

Neural network (NN) models are effective at function approximation, information classification, data filtering, and clustering. Because they were originally designed to mimic the human brain, neural networks can model user behavior. For instance, Billsus and Pazzani develop a neural network recommender for EachMovie dataset [23]. It models the recommendation process as a classification problem that separates favorable items from unfavorable ones. Experimental results show good accuracy and prove that neural networks may be used for recommendation purposes.

Neural networks can be trained to recognize rating patterns. In fact, trained models demonstrate rapid default reasoning, insensitivity to inconsistent input, and quick learning ability [36]. The greatest advantage of neural networks is their ability to stereotype users into categories that were not previously defined. This model can respond to novel input that is only slightly similar to previously learned examples. Such systems can handle partial and erroneous cues without making major errors. They can consistently associate an incomplete rating pattern with a complete test case that has a known opinion. We implement this model as one of our combination algorithms.

The ability of neural networks to recover complete data based on a partial key is called associative learning. We plan to use this property to predict unknown ratings by feeding a partial rating vector into the network and receiving a recommendation as output. Some researchers have already used Restricted Boltzmann Machines, a neural network with a fixed amount of hidden binary nodes, to make such recommendations [12, 83, 140]. The experiments on Netflix dataset show that a neural network based recommender can perform better than the SVD model.

Figure 3.2 shows a neural network segment for a single rating on a five point scale. Each discreet rating value is represented by a single excited binary node. A missing rating does not excite any input nodes. During the training stage, we feed known ratings into the model and expect known answers to come out. Otherwise, the back-propagation algorithm updates link weights in order to receive the desired output. The training continues until it converges to within an error threshold. To make a recommendation, we feed known ratings from an active vector and record the

NN output.

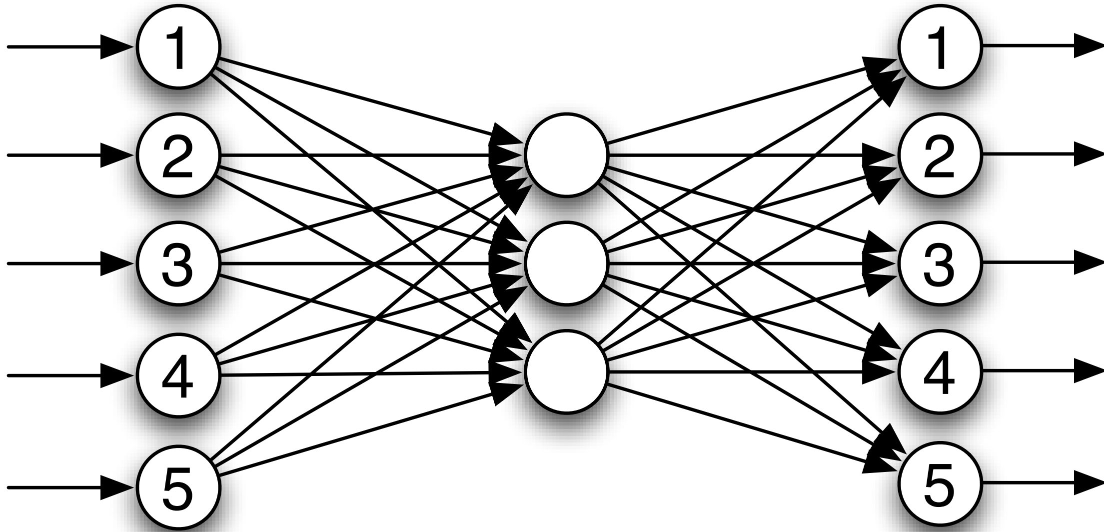


Figure 3.2: Prototype Neural Network Segment

More formally, the neural network training process may be described as follows. Each node of a network takes in a set of weighted inputs and produces a single value [158, 175]. To make sure it is within range, the net input is fed into a bounded activation function, $\sigma$. The formulas for computing the net input and output are as follows:

$$net = \sum_{i=0}^{n} w_i x_i$$

$$o = \sigma(net)$$

We use a sigmoid function because its range is (-1..1) and it has a convenient derivative, which allows for easier gradient descent on an estimation error surface [100, 114]. Below are the formulas for the original sigmoid function and its derivative:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

We can measure the error by comparing the desired output $t_d$ and the actual output $o_d$ over a set of training cases $D$:

$$E[\vec{w}] = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

We can now compute the gradient of this error with regards to individual weights. Due to the sigmoid function properties, we can further simplify the formula [100]:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$\frac{\partial E}{\partial w_i} = - \sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

Finally, we update link weights after each training case, $t$ [72]. The formula for computing the next weight of a particular link is as follows:

$$w_j(t+1) = w_j(t) + \eta o (1 - o) e_{i,j} x_{i,j}$$

Here, $\eta$ is the learning speed, $e_{i,j}$ is the error for that link, $o$ is the output of the node, and $x_{i,j}$ is the input sent over the link. The initial error may be observed on the network output directly. It is then propagated backwards and adjusted by the links' weights, i.e., strong links that transmit large errors should be adjusted more. As a result, the error of a particular node is a weighted sum of the errors of the nodes in the following layer.

## 3.2 Memory-Based Algorithms

Memory-based approach does not create or maintain a model. This method works directly with the dataset, inspecting it before each recommendation. As a result, the

most recent information is used as soon as it becomes available. However, because calculations are done on-demand, memory-based approaches are notoriously slow when applied to large datasets [44, 61]. In fact, input selection is the slowest and most crucial part of this algorithm. Fortunately, this approach requires the same amount of data for each recommendation [142]. Therefore, we can preprocess the input and improve performance of our prototype.

We focus this research on memory-based algorithms because they offer superior accuracy to many model-based recommenders [26, 31, 142]. Furthermore, this popular recommender approach is simple and intuitive, does not require many tuning parameters or long training sessions, and can justify recommendations [12], which is one of our future research directions. This approach may not be the fastest, but it is the most appropriate for improving recommendation accuracy in large and sparse datasets.

Memory-based recommenders identify a subset of all users, called neighbors, which are similar to the active user. Because of the neighborhood concept, this method is often called the K Nearest Neighbors (KNN) approach [83, 170]. Some versions of this approach view a dataset as a collection of user vectors with a specific number of dimensions, corresponding to items they rated [61, 143, 163]. However, item vectors are also possible [44, 94, 104]. In fact, previous research shows that the item-based approach produces more accurate results [70, 94, 142, 145, 148] because item vectors better capture the opinions of a neighborhood.

Our analysis of the Netflix dataset shows why item-oriented input is most accurate. Figures 3.3 and 3.4 show the vote counts for movies and users. In fact, 56% of all

movies have over one thousand ratings. An average movie has 5,654.5 ratings, which increases the possibility of rating overlap and provides more potential neighbors. Few users have this many ratings. In fact, half of the user population has 100 ratings or less, with an average user rating only 209.2 movies. With the exception of a few extremely active accounts, user vectors are usually sparse, which reduces the chance of finding similar neighbors.

Once the neighborhood is identified, the item-based KNN method combines items' ratings and their similarities to calculate a predicted rating for the active item. A more formal example of this algorithm is presented below. The mean vote for an item is defined as the sum of its ratings divided by their count. Here, $r_{u,i}$ is the vote submitted by a user $u$ on an item $i$ and $U_i$ is a set of users who rated the item $i$:

$$\bar{r}_i = \frac{\sum_{u \in U_i} r_{u,i}}{|U_i|}$$

The predicted vote for a user $u$ on active item $a$ is then a weighted sum of other items' ratings adjusted by their mean [26, 29]. In this formula, $n$ is the neighborhood size, $w(a, i)$ is the similarity of a neighbor $i$, and $k$ is a tuning coefficient:

$$P_{u,a} = \bar{r}_a + k \frac{\sum_{i=1}^{n} w(a,i)(r_{u,i} - \bar{r}_i)}{\sum_{i=1}^{n} w(a,i)}$$

An accurate similarity measure is an essential part of the KNN approach. In fact, it is often the distinguishing characteristic of an algorithm. In general, a recommender system considers vectors with similar ratings as neighbors. To quantify the similarity between any two vectors, it must first identify common dimensions, i.e., vote overlap.

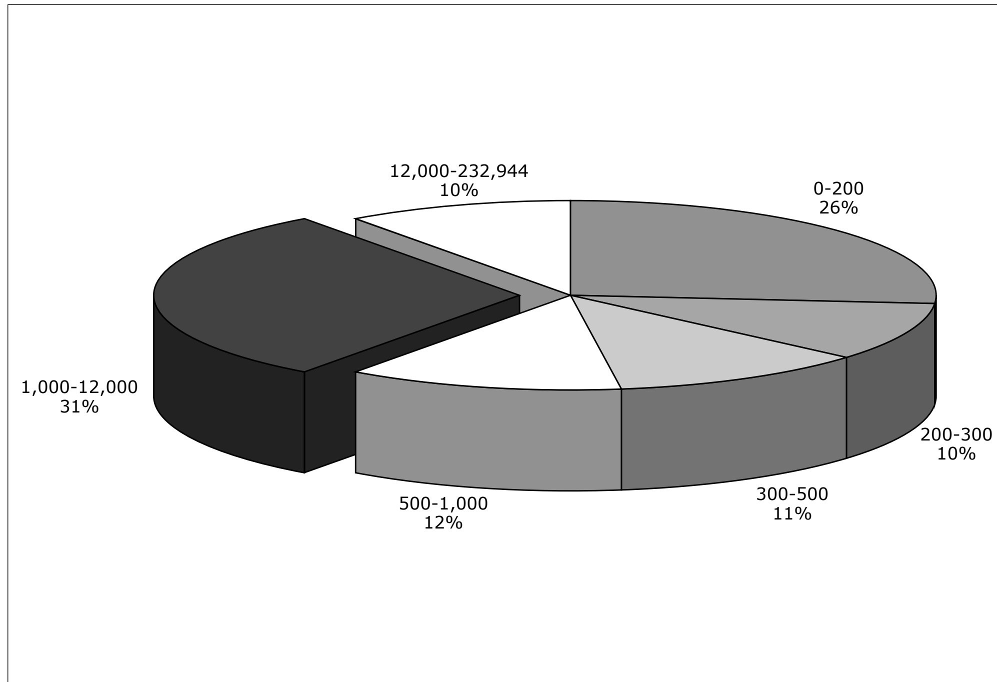


Figure 3.3: Netflix Movie Vote Count Distribution

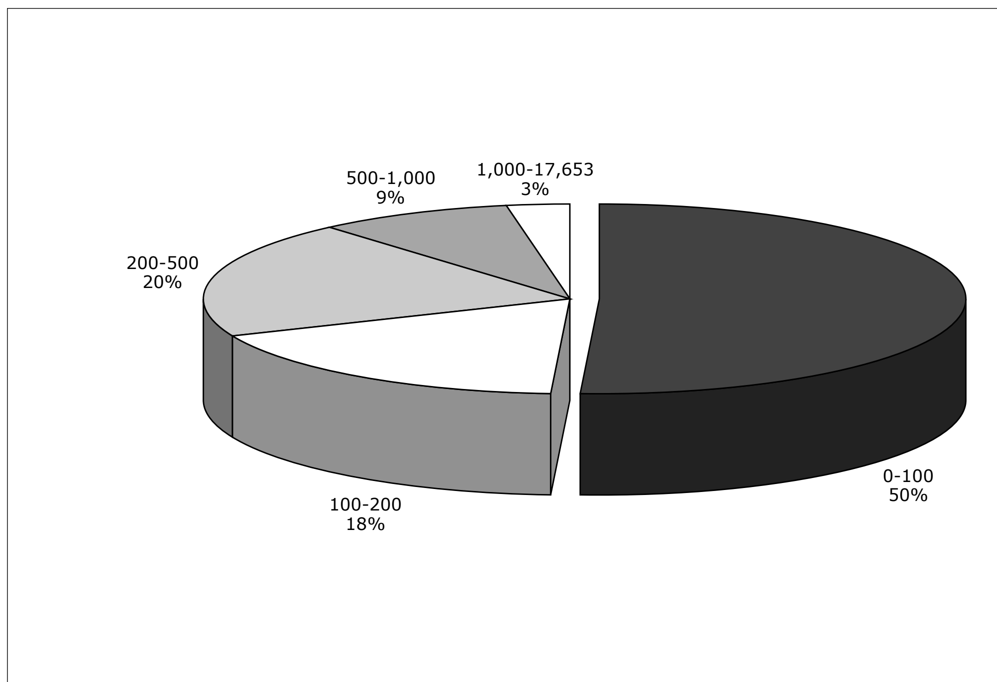


Figure 3.4: Netflix User Vote Count Distribution

Then it applies a similarity measure to the two sets of ratings to determine how close they are.

Two popular similarity measures are normalized Manhattan distance and normalized Euclidean distance. Normalized Manhattan distance is the mean absolute difference among commonly rated items between active user $a$ and its neighbor $u$, $i \in I_a \cap I_u$:

$$w(a,u) = \frac{\sum_{i \in I_a \cap I_u} \mid r_{a,i} - r_{u,i} \mid}{\mid i \in I_a \cap I_u \mid}$$

Normalized Euclidean distance is the Euclidean distance between the two user vectors divided by the amount of commonly rated items:

$$w(a,u) = \frac{\sqrt{\sum_{i \in I_a \cap I_u} (r_{a,i} - r_{u,i})^2}}{\mid i \in I_a \cap I_u \mid}$$

Both measures quantify the similarity of two vectors, but they do not consider the various rating scales that users may employ. These measures vary with vector magnitude, so two users with similar rating patterns could be considered different just because their vectors have different lengths. This could be problematic in large datasets with a wide variety of users and their individual rating scales.

Another popular measure that does not take vector length into account is cosine similarity. It compares two users by taking a cosine of the angle between their rating vectors [105, 143, 173]. It is the sum of products of commonly rated items divided by the product of vector lengths:

$$w(a,u) = \frac{\sum_{i \in I_a \cap I_u} r_{a,i} r_{u,i}}{\sqrt{\sum_{i \in I_a \cap I_u} r_{a,i}^2 \sum_{i \in I_a \cap I_u} r_{u,i}^2}}$$

The individual rating scales may be different, but users with similar preferences will have their rating vectors pointing in approximately the same direction. Cosine similarity is accurate because it considers the agreement among ratings to be more important than vector lengths [29, 142]. We consider this similarity measure in our empirical study.

The original cosine similarity measure is effective and has been widely used in information retrieval research. However, it does not explicitly adjust to users' rating scales [11]. For instance, a three star rating could mean "average" to one person and "good" to another. Standard cosine similarity uses actual ratings, so unless two users have the same rating scales, their similarity is lost. Alternatively, linear regression approximations can recognize similarity in such situations. For example, Pearson's correlation measures linear dependence between two sets of ratings:

$$w(a,u) = \frac{\sum\limits_{i \in I_a \cap I_u} (r_{a,i} - \bar{r}_a)(r_{u,i} - \bar{r}_u)}{\sqrt{\sum\limits_{i \in I_a \cap I_u} (r_{a,i} - \bar{r}_a)^2 \sum\limits_{i \in I_a \cap I_u} (r_{u,i} - \bar{r}_u)^2}}$$

This measure is similar to cosine similarity, except individual ratings are normalized by the vector average [13, 61]. This adjustment improves accuracy of even the most basic recommenders. In fact, it is most effective in sparse datasets, where linear regression is more easily established [15, 142]. We consider this similarity measure in our empirical study.

## 3.3 Conclusion

In this chapter, we discuss two types of collaborative filtering recommendations. Existing research shows that memory-based approach is superior to probabilistic and associative rule models. These models are too general to make personalized recommendations, whereas memory-based methods are simple, accurate, and use new data immediately. We focus the rest of this research on memory-based recommendations. However, our experiments include RSVD and NN models for comparison purposes. Since dataset size and sparsity are the primary reasons for memory-based algorithm deficiencies, we address them in the following chapters.

# Chapter 4

# The Proposed System Architecture

Current recommender systems do not allow flexibility, which is required for improved recommendation accuracy. They are usually implemented as proprietary applications with a particular dataset in mind. Many of them rely on a single combination algorithm to produce recommendations [65, 136, 148], so choosing a different algorithm at runtime requires multiple updates to a tightly coupled and inflexible software. Suggestions based on a consensus of algorithms are often more accurate than any individual approach [12, 14, 56, 107], but combining different recommendations is usually a post-processing task, which is slow and redundant. Therefore, studying recommendation accuracy in such environments is a cumbersome process.

To achieve desired flexibility, we divide our system into an input generation algorithm and multiple combination algorithms. This separates data management, the slowest and most demanding part of the system, into a dedicated component. It also simplifies the combination algorithms and eliminates redundant data processing previously required to combine recommendations. The nature of our research de-

fines the architecture of our system. We develop a simple model that can perform complex analysis of a large dataset and produce thousands of recommendations with slightly different parameters. Our architecture allows structured development and rapid experiments in the finished prototype.

The previous chapter described the ways of making a recommendation from a set of ratings. This and the following chapter focus on the system architecture that delivers such ratings and the input generation component that selects them from the dataset. The architecture explains the primary components of our system, their functionality, interactions among them, and the reasoning for such composition. We also formulate a detailed specification of the input generation algorithm with a set of desired properties. Finally, we discuss the satisfaction of our system requirements, which range from functional aspects to various non-functional constraints like performance, efficiency, scalability, adaptability, and ease of future development.

## 4.1 Distributed Computation with Accessible Data

We propose using the benefits of the network-centric software architecture to create a highly accessible, multi-platform recommender system. Network-centric systems consist of multiple computers working together to accomplish a common goal. In our case, they produce thousands of recommendations quickly and accurately. The network plays a central role in this approach because it helps synchronize the computing nodes. The network is the only thing these computers have in common, so there are no requirements for computer hardware, operating system, or software. Network-

centric systems will admit anyone, as long as they comply with the communication protocol, which results in great implementation flexibility.

Large network-centric systems have access to vast computing resources, which could be used to solve previously impossible problems. For example, the PocketLens recommender system uses a central server to instruct individual nodes to build their own models and compute recommendations [105]. In fact, its task is easily decomposable into individual recommendations to be computed by separate machines. A similar architecture is implemented in the SETI@Home project, where a screensaver software analyzes radio signals in search of extraterrestrial intelligence. This distributed system is considered one of the most powerful computers in the world and we adopt its architecture to achieve performance and scalability.

The implementation of a network-centric system usually involves developing an overlay network over the existing communication structure. In most cases, the individual processing components are connected via the Internet [110]. Therefore, low bandwidth and high latency systems would be unusable. We address the bandwidth requirement by reducing the amount of data that travels between a user and a server. We also decrease latency by preprocessing some of the data. As a result, we can produce fast recommendations while retaining the flexibility of a network-centric architecture.

We incorporate many best practices of the effective distributed systems. For example, we physically separate the software components that request services from the components that provide such services [156]. This allows us to scale the number of input generation instances, combination algorithm instances, or both. We also

make the service providers unaware of the requesters' identity [156]. This allows the input generation component to service many combination algorithms as if they were one. Finally, we insulate the requesters from one another [156]. Because combination algorithms depend only on the input, we can add, remove, or modify them without affecting the rest of the system. Note that these generic practices may apply to many different distributed architectures, but their rationale justifies the following design decisions.

## 4.2 Multi-Platform Implementation with Minimal Client Requirements

Client-server architecture is the most frequently used style for network applications. As the name suggests, it consists of two components: a server that offers a set of services and a client in need of such services [49]. This architecture is a perfect example of separation of concerns. Clients usually handle the user interface functionality, which keeps the server-side code simple. Servers perform intensive computations and store the data, so clients have minimal performance and storage requirements. The client-server separation also improves system modifiability as both components may be updated independently. We adopt this architecture to structure and simplify our development efforts.

The choice of client software has major consequences on the future performance of a recommender system. Distributed recommender systems usually employ smart clients to process and store recommendation data [105, 165, 166, 169]. Smart clients

| | |
|---:|---|
| Server Processor | Quad 3.2 Xeon EM64T |
| Server Memory | 16GB |
| Server Operating System | Redhat Linux x86_64 |
| Database Server | MySQL 5.0.22 |
| Web Server | Apache 2.2.16 |
| Web Client | Chrome 8.0.552 |

Table 4.1: Prototype Software and Hardware Configuration

offer more computational performance, but require custom software to be installed on each computer, which limits deployment options. We use existing Web client software, which has limited processing and storage abilities. However, we design our system such that the clients perform only lightweight computations on minuscule amounts of data. As a result, our prototype can produce recommendations on virtually any Internet-ready device, regardless of the processor architecture or operating system.

Building our system on top of an existing infrastructure helps us focus on the main purpose of this research without sacrificing performance, robustness, or reliability. Fortunately, there are many reliable implementations for both client and server software, e.g., Apache server and Chrome client. We outline details of our server hardware and software configuration in Table 4.1. We use the latest stable software to manage the dataset and access it through the Web. Furthermore, our system can work with any Web client, but we use Google Chrome because it offers the fastest client-side performance.

Some quality attributes promoted by the client-server architecture include scalability, availability, and performance. Our system is scalable because additional clients can easily join and contribute their recommendations. It remains available even if a

client quits during the recommendation process. The reserved recommendation will be unprocessed, but all remaining suggestions will be completed by other clients. Finally, our system allows multiple clients to make concurrent recommendations. This quality is especially beneficial for our case study, where we evaluate the recommendation accuracy of multiple algorithm configurations over thousands of recommendations.

## 4.3 Complexity Management with Layers

One of the most popular ways to organize a complex software system is to break it down into layers, because their strict communication rules encourage solid software design. This architecture reduces coupling, as the communication for a layer is usually restricted exclusively to the layers adjacent to it [54]. Loosely coupled components may be implemented and tested independently, which is a desirable quality for our system. Layered architecture also improves component cohesiveness as each layer usually contains functionality related to a particular level of abstraction [54, 159]. Layers force us to group similar functionality together, so any issues are more likely to be contained within a single layer. We use this architecture style to visualize our system and build it one layer at a time.

We divide the business logic between a server and a set of clients, unlike a traditional approach where all functionality resides on the server side. Figure 4.1 shows the logical and physical structure of our system. The figure also shows a dependency that clients have on servers, i.e., a client may not generate a recommendation unless it has a server to provide it with input data. This architecture implements all three best

practices of an effective distributed system, the components are physically separate, the clients appear indistinguishable to the server, and clients are unrelated to each other.

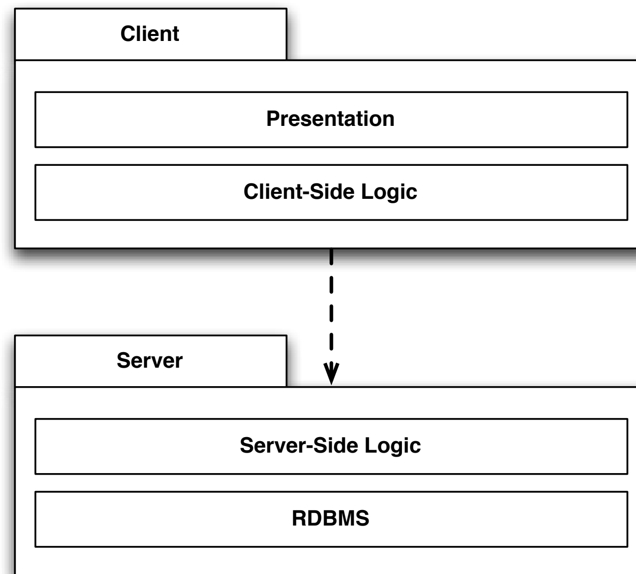


Figure 4.1: Layered Client-Server Architecture

Starting from the top of our stack, the presentation layer interacts with a user directly. It displays each recommendation and helps request new ones. As far as the user is concerned, the entire system is abstracted by this layer. It contains no business logic, so it can change radically, without affecting the rest of the system [132]. Such flexibility allows us to easily change the “skin” of our system or incorporate it into an existing application.

Below, a traditional application layer is broken down by two tiers into client-side logic and server-side logic layers. Together, they are responsible for extracting, optimizing, transmitting, and processing the dataset into a recommendation. The processor-intensive data manipulation functionality resides on the server side, next

to the data. Lightweight processing resides on the client side and has nothing to do with the original dataset. Its sole purpose is to combine a small matrix of ratings into a single recommendation. The performance improvement of such separation might be negligible for individual recommendations, but it allows us to produce multiple types of recommendations simultaneously. For example, the server could send the same input to a KNN client and an RSVD client to receive two recommendations in the time it takes to generate one.

The bottom layer is reserved for a relational database management system (RDBMS). Its responsibility is to store and retrieve ratings from the dataset. This layer contains no business logic either, which keeps the coupling down. It also abstracts any load balancing, caching, or clustering that a database management system may employ [34, 60]. Managing data in a separate layer is beneficial because we can use an existing database management system. There are faster ways to access the dataset, but we prefer the convenience and reliability of a relational database over the speed of a proprietary data management system. In fact, standard database connectivity is often preferable to custom data structures and access methods [54]. We use the MySQL database because it lets us focus on the main goal of this research rather than on optimizing data access. Once a satisfactory solution has been developed, we plan to improve its performance by implementing it in a more optimized environment.

## 4.4 System Adaptability with Strict Interfaces

The flow of data and control through the layers defines our recommendation process. Figure 4.2 breaks it down into distinct steps. It starts with a *recommend* function call at the presentation layer in Step 1. The user may choose to provide a user and item ID for a desired recommendation. However, since both parameters are optional, the system may automatically choose an input matrix that has not yet been processed. The recommendation request is then encoded into an HTTP packet and sent to the server. During Step 2, the request is parsed, validated, and populated with any necessary lookups. At this step, a specific user and item ID must be established because the recommendation process may only continue when it is centered around a single rating by a known user on a known item.

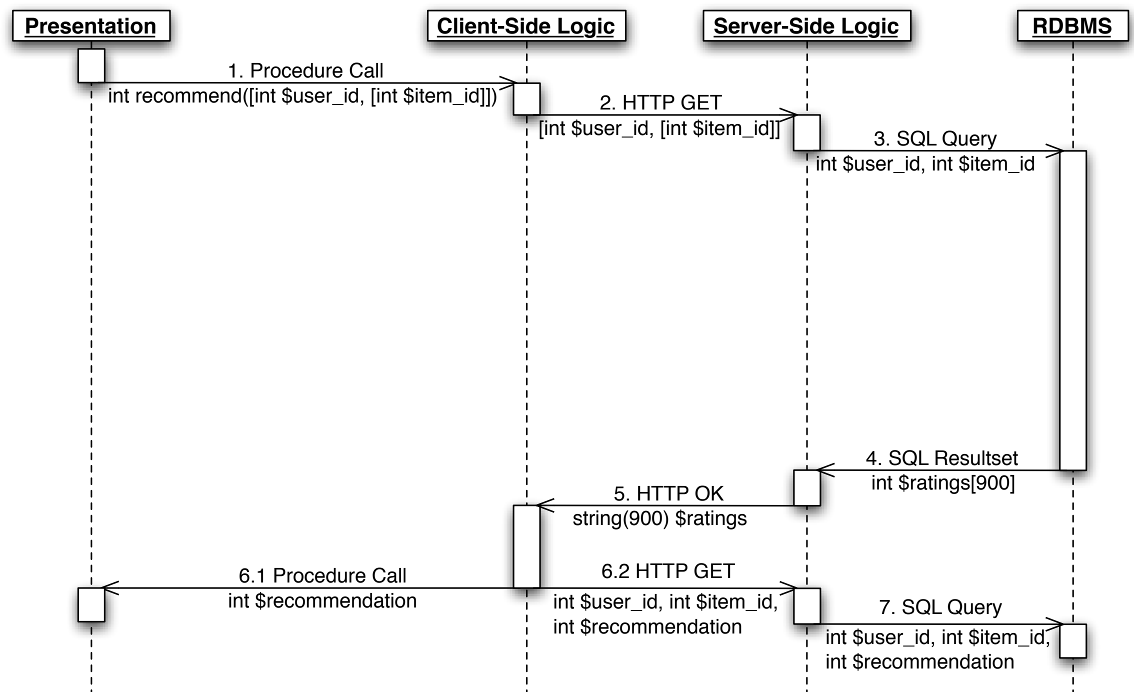


Figure 4.2: Layer Interaction During a Recommendation Process

Step 3 of the recommendation process is time-consuming because the server-side logic must process a large amount of data. It is also a place where most performance enhancements occur. For instance, if the dataset is partitioned across multiple machines, each one of them will receive the same query and their results will be combined. Furthermore, the dataset may be distributed across several hard drives and multiple virtual tables to improve response time [34, 60]. Regardless of the performance considerations, the RDBMS layer returns a set of ratings in Step 4.

These ratings are organized in a string, which is then sent to the client in Step 5. Once the input matrix reaches the client, the data inside it gets combined into a suggestion that is presented to the user or sent back to the server in Step 6. The input matrix is bound to 30 user and item vectors, which makes the largest possible matrix of 900 characters. The size of the data moving from server to client is extremely small, which makes it ideal for slow networks as well as lightweight combination algorithms.

To reduce coupling, layer communication occurs only through explicit, public interfaces. For example, the RDBMS layer does not expose its internal state other than via operations intended to modify that state in server-side logic layer [156]. As a result, combination algorithms are completely oblivious to the input generation process, what algorithm was used to perform it, or what database the data came from.

## 4.5 Future Extensibility with Prebuilt Components

The component-based architecture is very similar to the layered approach in a sense that each part of the system serves a single purpose. The main idea behind this style is that complex systems may be constructed by assembling a number of simple building blocks, abstracting implementation details and separating the architect and developer concerns [42, 51, 52]. However, architectural mismatch is a major problem with component-based development. In fact, the effort to join the different components usually exceeds that of writing the entire system from scratch.

We address this issue by describing a strict communication interface, thus removing any ambiguity regarding our component interaction. Figure 4.3 shows the two main components of the system, input generation algorithm and multiple combination algorithms. Each combination algorithm understands the *get_matrix* and *recommend* interfaces. The *make_matrix* interface has two required parameters (*user_id* and *item_id*) and returns a newline-separated string of item ratings. The *recommend* interface accepts this string and returns a single numerical recommendation.

One quality we are particularly interested in is the ability for different implementations of the same component to be used interchangeably. In fact, the flexibility of our prototype comes from replacing combination algorithm implementations [159]. Likewise, the input generation component is compatible with any collaborative filtering combination algorithm. If a recommender can work with a dataset of ratings, it can work with our input generation component, because our algorithm produces a matrix that is essentially a smaller version of the dataset.

Our architecture simplifies future recommender system development. A large

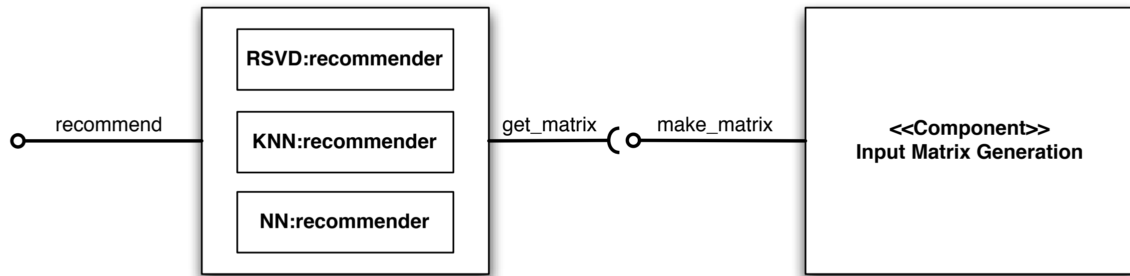


Figure 4.3: Proposed Component Communication Interfaces

number of existing systems use similar functionality for the data management part of the recommendation process [61]. Since there is not much variability in the way the ratings move from the dataset into the combination algorithm, it would be beneficial to include all of the related functionality into a single unit. Our input generation component represents a considerable portion of the entire system, so new recommenders may be constructed quickly.

Our architecture also makes it possible for the prototype to become a commercial product. However, it does not have to be Web-based. Our data management component coupled with a combination algorithm could form a standalone unit to be distributed across a number of processing nodes. As long as each node has a copy of the dataset, all recommendations could then be combined and used in a business model or presented to users. In fact, the entire recommender system could be abstracted by a load-balanced cluster of high-performance computers that populate a single table of recommendations.

## 4.6 System Dataflow Bottlenecks

We model recommender input as a matrix where items are rows and users are columns. Each cell contains a rating that is associated with a single user and an item. Figure 4.4 shows our prototype data flow. The input generation component locates all relevant ratings and chooses the best ones for the input matrix. The matrix is fed into a combination algorithm that, as the name implies, combines ratings to produce a recommendation.

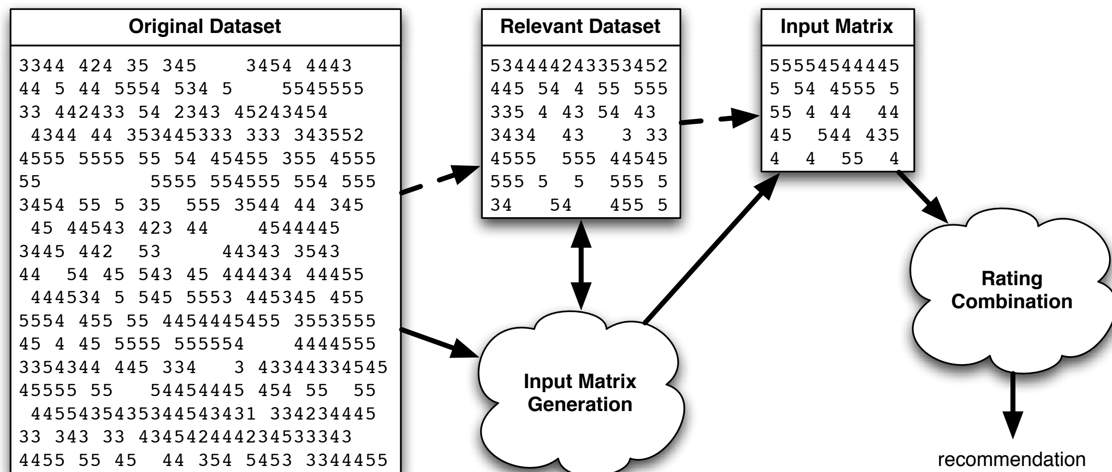


Figure 4.4: Proposed System Data Flow

To generate an input matrix, we first identify all relevant ratings on items rated by the active user and by users who have rated the active item. For example, if making an input matrix for Alice on Titanic, we consider all movies Alice rated and all users who rated Titanic. This data access task is time-consuming because a large portion of the dataset may be relevant. However, the relevant dataset may contain opinions that are not as valuable as others. Therefore, the input matrix generation algorithm chooses only the most valuable ratings and places them into the input

matrix. This data processing task is also time-consuming because all relevant ratings must be sorted.

Different matrices take a different amount of time to produce. To standardize the time it takes to receive the recommender input, we employ multiple replicas of the input generation algorithm to preprocess the data. They produce consistently sized, small, and highly valuable input matrices that enable quick and accurate recommendations in a fixed amount of time, regardless of the user/item popularity. To generate them efficiently, we must consider the communication, computation, and space utilization of our approach.

### 4.6.1 Efficient Communication Utilization

The more layers a system has, the slower it operates. All communications must pass through the layers linearly, which means that middle layers must be involved during the request and response phase of each recommendation. A finished recommendation does not have to travel through all layers, including the HTTP stack on top of which our system is built. Since no layers may be skipped, the performance of our system may suffer [54,132,159]. However, keeping the layers unaware of each other outweighs the performance benefits of passing data directly to the database, because the burden of submitting a single rating is negligible.

We employ a multi-layer architecture for its customization properties, but some layers may be removed to improve performance. To make sure that our system can adapt to such a change, we keep only interaction facilities inside connectors. In other words, the HTTP and MySQL connectors contain no logic, but rather the means

of transmitting requests and their responses. For example, we can implement our recommender system on a different database management system by replacing the MySQL connection. Likewise, we can change how a matrix travels to a combination algorithm, so when we replace the HTTP connector, the system remains operational.

For example, Figure 4.5 shows an embedded recommender system with no Web interface. It eliminates the overhead of two additional layers by sending the recommendation directly to the database. Such configuration exhibits the most optimal use of the communication medium, but it is more difficult to change algorithms and their configurations at runtime. Therefore, we reserve this configuration for production systems and implement a more customizable architecture for our case study. However, we use a version of this approach to precompute input matrices.

Our client-server responsibilities are guided by communication efficiency. We keep frequently interacting components close because latency affects interaction efficiency [156]. For example, it is inefficient for a combination algorithm to communicate with a remote dataset. If the input generation component resides on the client side, a single recommendation will produce a large amount of network traffic, which would decrease performance and limit scalability. Therefore, we limit the amount of information that goes through a slow network connection and abstract the data with a dedicated component.

The input generation component needs frequent and resource-intensive access to the dataset. We place it on the same host as the database, with a direct connection to the data. Our server has a more powerful processor and more memory than an average personal computer. Therefore, we use its resources to perform the most

intensive operations, i.e., data management.

If we could not physically separate the two components, we would have to use a slow connection to do a time-consuming task or we would have to sacrifice modifiability. Our proposed architecture accomplishes several things at once, we can modify combination algorithms independently of the recommendation process, their interactions with the server are minimal, and all data intensive processing occurs on a dedicated machine with the best available resources and the most optimal data access.

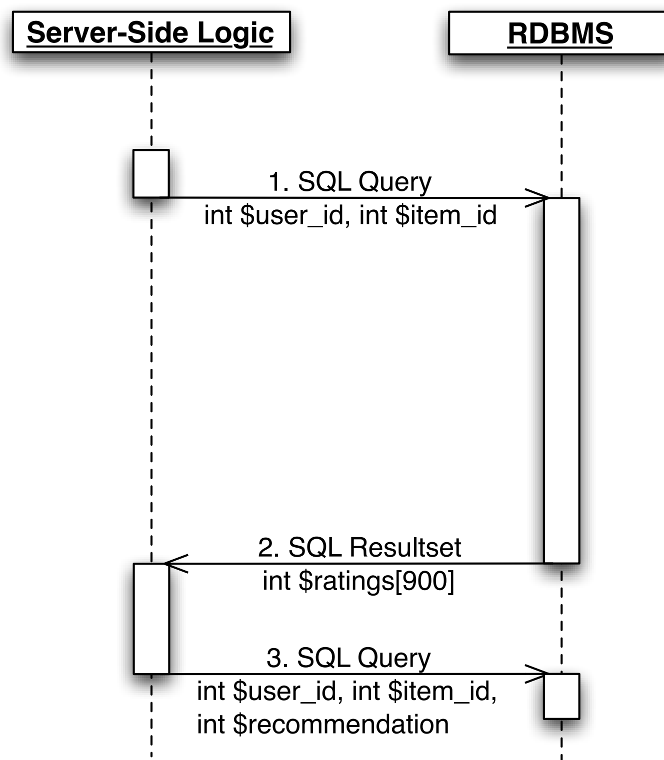


Figure 4.5: Recommendation Process with an Embedded Combination Algorithm

### 4.6.2 Efficient Computation Utilization

Previous configurations show a more linear approach to making recommendations. They provide a logical organization for a sequence of events that must occur before a user receives a suggestion. However, previous discussion also identifies the two bottlenecks in the recommendation process, data access and data processing. To improve online performance, these time-consuming tasks may be performed offline. If we precompute the data necessary to make a recommendation, we can improve recommendation performance, producing thousands of recommendations in minutes instead of hours.

We model the server-side code such that the resource-intensive data processing functionality is contained in a standalone unit available only through a strict interface. This allows us to keep the most complex and slowest part of the system separate from everything else. Additionally, we do not want the slowest component to dictate the overall performance. In other words, we want to prevent the combination algorithms from waiting idly for their input.

We maximize computational resource utilization with multiple instances of the input generation algorithm. They synchronize through a shared database table, which adds to the complexity of the system. However, we limit the amount of information shared among input generation instances and enforce local computation within each instance. The resulting architecture can efficiently use all of the available server resources to quickly preprocess the recommender input.

Our architecture also allows efficient evaluation of multiple recommendation approaches due to a dedicated input generation component. Each recommendation is

based on the same input data, so the most resource-intensive task is not performed redundantly. Furthermore, because combination algorithms are decoupled from the input generation algorithm, multiple clients may produce different kinds of recommendations simultaneously. Likewise, multiple input generation processes may work concurrently to generate input matrices. Both recommendation and input generation tasks lend themselves to distribution because their responsibilities are independent and clearly defined. Therefore, we replicate both components to scale our system.

### 4.6.3 Efficient Space Utilization

Caching input matrices increases our memory footprint, but it also results in a faster system. We address the space utilization issue by separating the data from the meta-data, which reduces the system's memory footprint and makes it more efficient. In other words, every time we make a recommendation it is not necessary to know the movie title, its genre, or release date. Such data may be useful for presentation purposes, but it is completely irrelevant in the input generation component. The data structure of our system represents the lowest common denominator of any collaborative filtering system. Therefore, we can accommodate a variety of datasets by adjusting the meta-data, which does not participate in the recommendation process, but does enhance the presentation.

We design our recommender system to maintain a major advantage of collaborative filtering, i.e., its ability to make suggestions on virtually any data. We model users, items, and numerical ratings as separate entities. Their relationships are fixed and are unlikely to change in other domains as long as the user, item, and rating

information is available. Our generic naming convention as well as the simplicity of the collaborative filtering approach effectively guarantee that our research may apply to a great number of domains.

## 4.7 Input Generation System Specification

This section presents a more detailed view of the input generation component. We describe it with Acme, an Architecture Description Language (ADL) developed at Carnegie Mellon University. It is a simple and generic language that is based on the premise that there is sufficient commonality among other ADLs. Acme embodies these commonalities while also allowing ADL-specific details [53]. Therefore, our Acme specification may be easily converted to a broad variety of other ADLs.

The core ontology of Acme includes components, connectors, systems, ports, and roles. Components represent the primary computations elements and data stores of a system. They correspond to the boxes in box-and-line descriptions. Connectors represent interactions among components. They mediate the communication among components and correspond to the lines in box-and-line descriptions [53]. In our case, connectors are the SQL links between the database and the functions. Component interfaces are defined by a set of ports. Each port identifies a point of interaction between the component and its environment. Connectors also have interfaces that are defined by roles. Each role defines a participant of the interaction represented by the connector [53]. Finally, systems represent configurations of components and connectors. The rest of this section describes the input generation system in terms

of these structural elements.

First, we define three types of roles in our system: provider, receiver, and controller (Listing 4.1). The idea is that the provider points to the source of the data, the receiver points to its destination, and the controller points to the component specifying which data to move. Note that all logic resides in the controller component, which ensures future adaptability. Each role has two rules that ensure that the role is attached to a matching port and that such attachment is unique. These constraints prevent the same connection from being used for different purposes, thus enforcing system consistency.

Listing 4.1: Role Types of the Input Generation Algorithm

```
1  Role Type r_provider=
2  {
3      rule CountCheck=invariant size(self.ATTACHEDPORTS) == 1;
4      rule TypeCheck=invariant forall r:Role in self.ATTACHEDPORTS |
5          declaresType(r, p_provide);
6  }
7
8  Role Type r_receiver=
9  {
10     rule CountCheck=invariant size(self.ATTACHEDPORTS) == 1;
11     rule TypeCheck=invariant forall r:Role in self.ATTACHEDPORTS |
12         declaresType(r, p_receive);
13 }
14
15 Role Type r_controller=
16 {
17     rule CountCheck=invariant size(self.ATTACHEDPORTS) == 1;
18     rule TypeCheck=invariant forall r:Role in self.ATTACHEDPORTS |
19         declaresType(r, p_control);
20 }
```

We use different connectors for control and data exchange. Our components interact through shared data access, which is a very efficient way to exchange large amounts of information. Such interaction is asynchronous, or distributed in time, which means that we need to establish the order in which writers and readers access shared data. The thread controller uses procedure calls to invoke the functions in a

predefined order, documented in Figure 4.6. Procedure calls synchronously exchange control between procedures and clearly illustrate the interaction paths among the system's components. We choose specialized connectors for exchanging data and control because they make our system efficient.

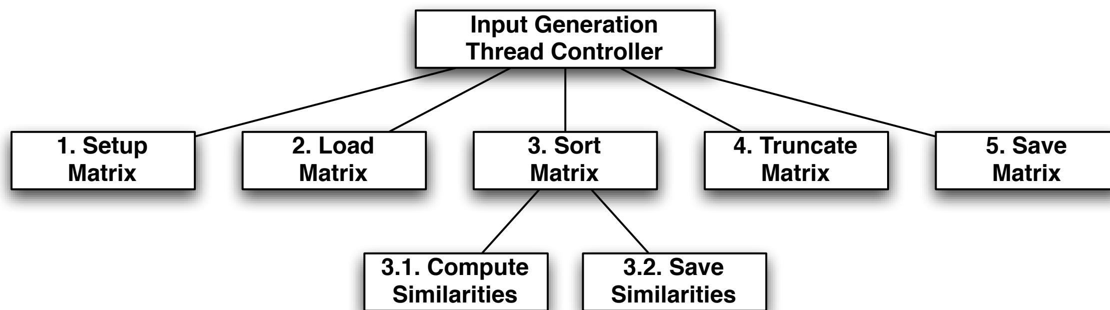


Figure 4.6: Input Generation Algorithm Order of Execution

We use two types of shared data connectors, data access and data moving (Listing 4.2). The access connector links a provider and controller components that interact among themselves, but the data they exchange remains in the provider component. The moving connector is the same, except it also links the receiver component. The controller interacts with the provider, but the resulting data is sent to the receiver component. Both connectors contain rules that ensure that each connection has appropriately typed roles, that there is a correct number of roles, that there are no two roles of the same type, and that the connecter has no dangling roles.

We also define three types of ports that a component may have: provide, receive, or control (Listing 4.3). Each port name describes whether the component provides, receives, or controls the data exchange. Each port contains two rules that ensure that it is connected to a properly typed connection role. Note that a single provide or use

Listing 4.2: Connector Types of the Input Generation Algorithm

```
Connector Type DataAccessConnT=
{
        Role controller:r_controller=new r_controller
        Role provider:r_provider=new r_provider

        rule TypeCheck=invariant forall r:Role in self.ROLES |
            exists t in {r_provider, r_controller} declaresType(r, t);

        rule CountCheck=invariant size(self.ROLES) == 2;

        rule UniqueTypeCheck=invariant forall r1:Role in self.ROLES |
            forall r2:Role in self.ROLES |
                r1 != r2 <->  ! exists t in {r_provider, r_controller} |
                    declaresType(r1, t) AND declaresType(r2, t);

        rule NoDangling=invariant forall r:Role in self.ROLES attachedOrBound(r);
}

Connector Type DataMovingConnT=
{
        Role receiver:r_receiver=new r_receiver
        Role provider:r_provider=new r_provider
        Role controller:r_controller=new r_controller

        rule TypeCheck=invariant forall r in self.ROLES |
            exists t in {r_provider, r_receiver, r_controller} declaresType(r, t);

        rule CountCheck=invariant size(self.ROLES) == 3;

        rule UniqueTypeCheck=invariant forall r1:Role in self.ROLES |
            forall r2:Role in self.ROLES |
                r1 != r2 <->  ! exists t in {r_provider, r_receiver, r_controller} |
                    declaresType(r1, t) AND declaresType(r2, t);

        rule NoDangling=invariant forall r:Role in self.ROLES attachedOrBound(r);
}
```

port may have multiple connections to accommodate different interaction with the same data. However, a single control port may only be used for a single purpose. This rule ensures that each computational component is dedicated to a single purpose.

Our system has two types of components, data controller and shared data (Listing 4.4). Because each component type has a different purpose, each type contains rules that prevent us from accidentally declaring the wrong components. The controller component must provide exactly one port of type control. The data component must provide at least one port of either provide or receive type, but there may not be

Listing 4.3: Port Types of the Input Generation Algorithm

```
1  Port Type p_provide=
2  {
3      rule CountCheck=invariant size(self.ATTACHEDROLES) >= 1;
4      rule TypeCheck=invariant forall r:Role in self.ATTACHEDROLES |
5          declaresType(r, r_provider);
6  }
7
8  Port Type p_receive=
9  {
10     rule CountCheck=invariant size(self.ATTACHEDROLES) >= 1;
11     rule TypeCheck=invariant forall r:Role in self.ATTACHEDROLES |
12         declaresType(r, r_receiver);
13 }
14
15 Port Type p_control=
16 {
17     rule CountCheck=invariant size(self.ATTACHEDROLES) == 1;
18     rule TypeCheck=invariant forall r:Role in self.ATTACHEDROLES |
19         declaresType(r, r_controller);
20 }
```

two ports of the same type. Note that we explicitly state that each component must provide a port, thus eliminating a possibility of a component with missing or unknown interface.

Listing 4.5 shows the entire system with connected components. The graphical view of this system is available in Figure 4.7. We have two shared data components and four data controllers. Note that the temporary data component is shared only among the functions of the active input generation instance, but the permanent data component is also shared among all instances of this algorithm. All components are connected with four connectors, two for data access and two for data moving. The system also contains rules that enforce its correctness. There must be at least one component and all components must be either controllers or data. Likewise, there must be at least one connector and all connectors must be either access or moving. Finally, port and connector rules ensure everything is connected and their type checks ensure that connections are properly aligned. For example, the number of

Listing 4.4: Component Types of the Input Generation Algorithm

```
Component Type DataControllerT=
{
    Port control:p_control=new p_control

    rule CountCheck=invariant size(self.PORTS) == 1;
    rule TypeCheck=invariant forall p:Port in self.PORTS declaresType(p, p_control);
    rule NoDangling=invariant forall p:Port in self.PORTS attachedOrBound(p);
}

Component Type SharedDataT=
{
    Port provide:p_provide=new p_provide
    Port receive:p_receive=new p_receive

    rule CountCheck=invariant size(self.PORTS) >= 1;

    rule TypeCheck=invariant forall p:Port in self.PORTS |
        exists t in {p_provide, p_receive} declaresType(p, t);

    rule NoDangling=invariant forall p:Port in self.PORTS attachedOrBound(p);

    rule UniqueTypeCheck=invariant forall p1:Port in self.PORTS |
        forall p2:Port in self.PORTS |
            p1 != p2 <->  ! exists t in {p_provide, p_receive} |
                declaresType(p1, t) AND declaresType(p2, t);
}
```

controllers should match the number of connections because otherwise the *r_controller* or *p_control* rules will be violated.

## 4.8 Conclusion

This chapter summarizes the architectural properties of our recommender prototype. Classic solutions lack the flexibility of the proposed architecture. Therefore, we describe applicable architectural patterns and show how they enhance fast and accurate experiments. We also document the communication lines between different components of our system and identify performance bottlenecks. Our main contribution is the separation of common functionality among multiple recommender approaches into a single component. We describe this module because it is the most important

part of our recommender system. We do not consider the internal behavior of the components yet, thus abstracting the complexity away. Such an architecture view should simplify further system design and aid its development.

Listing 4.5: Input Generation System Specification

```
1  System InputGenerationInstance=
2  {
3      Component PermanentSchema:SharedDataT=new SharedDataT
4      Component TemporarySchema:SharedDataT=new SharedDataT
5
6      Component LoadMatrix:DataControllerT=new DataControllerT
7      Component SaveMatrix:DataControllerT=new DataControllerT
8      Component SortMatrix:DataControllerT=new DataControllerT
9      Component TruncateMatrix:DataControllerT=new DataControllerT
10
11     Connector Conn0:DataMovingConnT=new DataMovingConnT
12     Connector Conn1:DataAccessConnT=new DataAccessConnT
13     Connector Conn2:DataAccessConnT=new DataAccessConnT
14     Connector Conn3:DataMovingConnT=new DataMovingConnT
15
16     Attachment TemporarySchema.receive to Conn0.receiver;
17     Attachment PermanentSchema.provide to Conn0.provider;
18     Attachment LoadMatrix.control to Conn0.controller;
19
20     Attachment SortMatrix.control to Conn1.controller;
21     Attachment TemporarySchema.provide to Conn1.provider;
22
23     Attachment TruncateMatrix.control to Conn2.controller;
24     Attachment TemporarySchema.provide to Conn2.provider;
25
26     Attachment PermanentSchema.receive to Conn3.receiver;
27     Attachment TemporarySchema.provide to Conn3.provider;
28     Attachment SaveMatrix.control to Conn3.controller;
29
30
31     rule ComponentCountCheck=invariant size(self.COMPONENTS) >= 1;
32     rule ConnectorCountCheck=invariant size(self.CONNECTORS) >= 1;
33
34     rule ComponentTypeCheck=invariant forall c:Component in self.COMPONENTS |
35         exists t in {SharedDataT, DataControllerT} declaresType(c, t);
36     rule ConnectorTypeCheck=invariant forall c:Connector in self.CONNECTORS |
37         exists t in {DataMovingConnT, DataAccessConnT} declaresType(c, t);
38 }
```

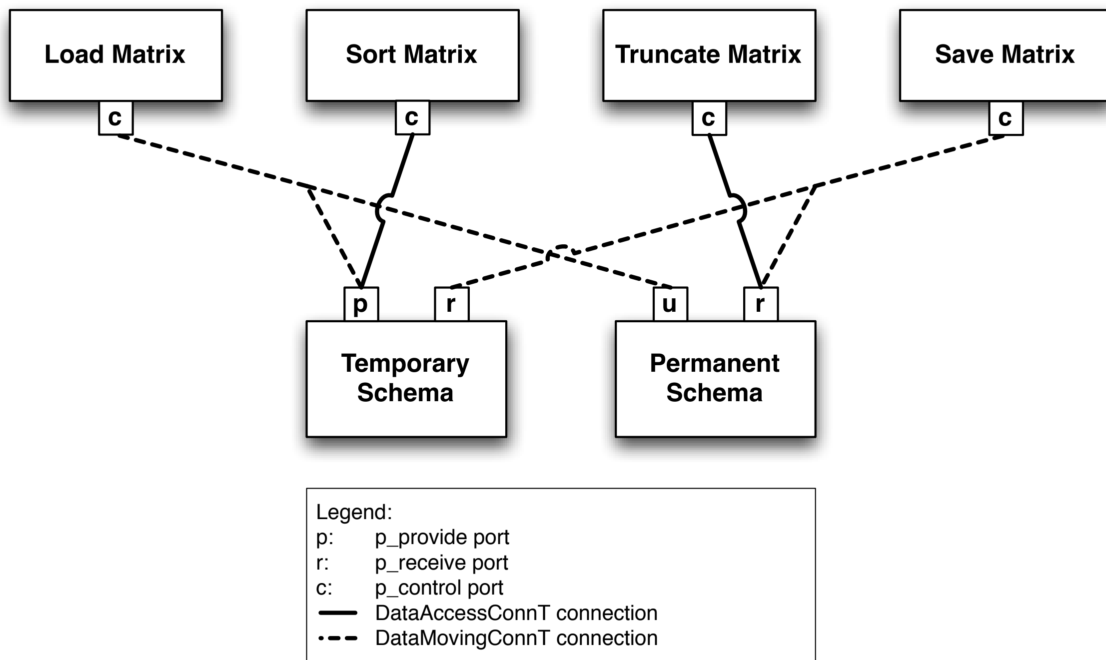


Figure 4.7: Input Generation Architecture in Acme ADL

# Chapter 5

# Improving the Quality of Recommender Input

Many effective combination algorithms are based on existing statistical, mathematical, or data mining principles. Some research focused on adjusting these formulas for improved accuracy [13,73,142]. However, most approaches assume that the amount of ratings is the sole reason for poor recommendation quality. We suggest that it is not the amount, but rather the relevance of ratings, that determines recommendation accuracy. In fact, many studies show dramatic recommendation quality improvements due to changes in the input data [29, 63, 146]. In a sense, optimizing the combination algorithm is comparable to fixing the symptoms, while improving the input data is addressing the root of a problem.

This chapter provides a more in-depth view of our data management component. We consider two different ways of establishing vector similarity within a dataset, which is a crucial step in the process of making collaborative filtering recommenda-

tions. First, we offer an informal conjecture as to why our way of locating similar neighbors is better than existing approaches. Then, we demonstrate the essence of our hypothesis through a simulation. We consider the different processes involved in making input matrices and justify their organization. Finally, we examine the statistical implications of our approach to show how it reduces the effects of random variability in the data, which leads to more accurate recommendations.

## 5.1 The Standard Input Generation Approach

Because our dataset contains millions of records, combining all relevant ratings is not feasible. Instead, we make recommendations from a subset of the most relevant ratings. Figure 5.1 shows how we identify the users and items that comprise an input matrix. This process consists of a series of simple operations, many of which could be executed in parallel to increase performance. However, there are two synchronization points in this workflow, when only one operation is running. Therefore, we break it down in two tasks at the synchronization boundary. This division is not necessary in a production system, but it helps us separate a strictly data access task from a strictly data processing task.

The standard input generation approach creates a larger matrix first and then reduces its size. This may be described in four steps. We start with a single cell matrix for the active user and item. Figure 5.2 shows the initial state of an input matrix. In this case, we are predicting a rating by user U1 on item I1. This step makes sure that the active user and item vectors are in the matrix, regardless of

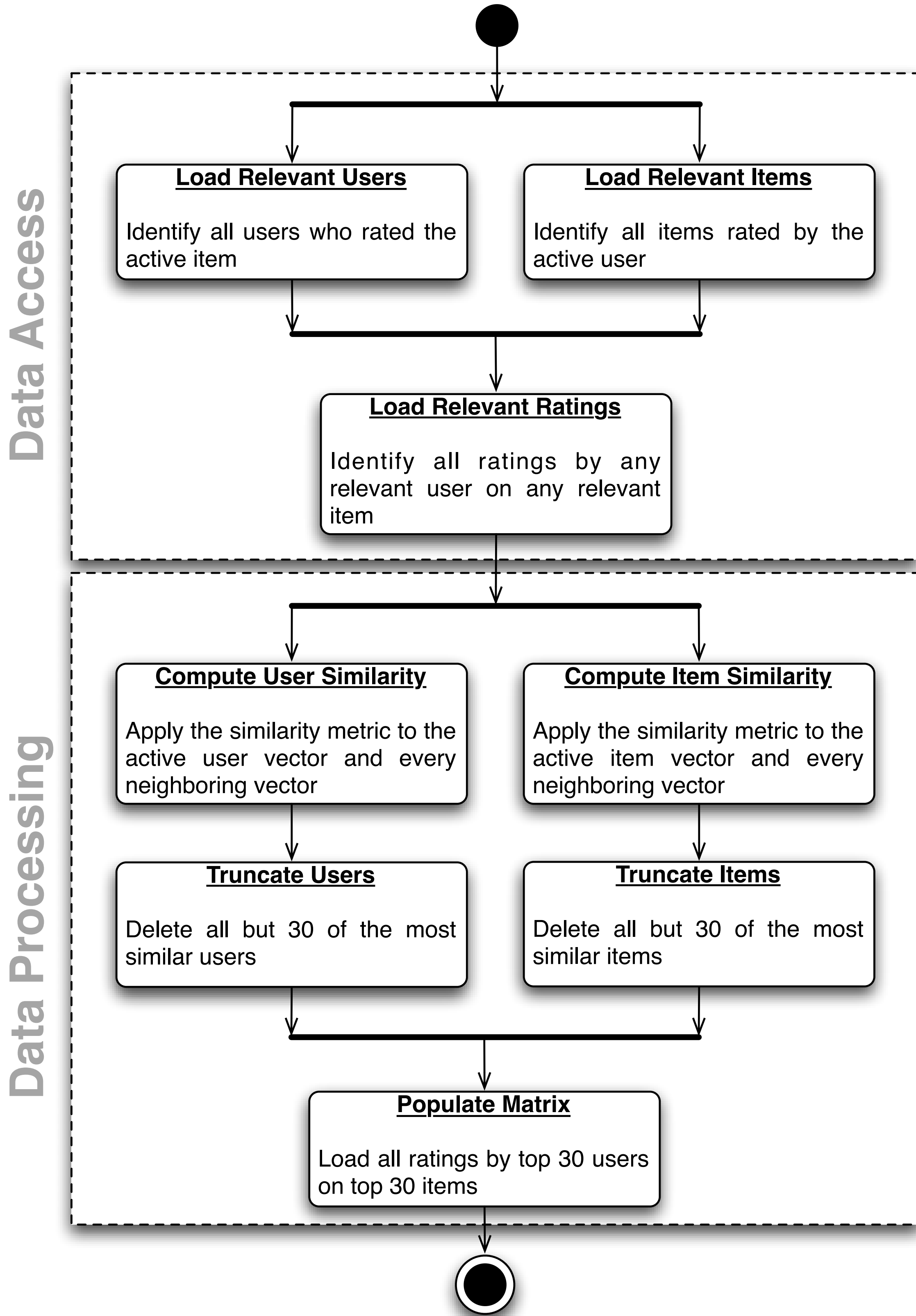


Figure 5.1: Input Generation Process Activities

whether we know the active rating. If the active rating is unknown, the two vectors would be an exception to the following step.

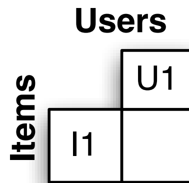


Figure 5.2: Input Matrix Generation – Initial State

The second step populates the matrix with all relevant user and item vectors as well as any ratings they have. At this point, columns are all users who rated the active item and rows are all items that the active user rated. Figure 5.3 shows the populated matrix. In this case, the active user has rated five items, and five other users rated the active item. Note that the first row and the first column of the input matrix always have values, with the exception of the rating we are trying to predict. This happens because the active vectors define the shape of the matrix, so a particular dimension is not considered unless it occurs within one of the active vectors.

The next step sorts columns and rows, while preserving rating association. In other words, rearranging the movies does not affect users' opinions about them. A variety of similarity metrics may establish the sort order. We use the cosine similarity and Pearson's correlation measures to move similar vectors closer to the active vector. Figures 5.4 and 5.5 show the third step of the input generation process.

Finally, we remove the least similar rows and columns. Figure 5.6 shows the final truncated matrix, where we keep the top three vectors. Since all neighbors are organized in order of decreasing similarity from the active vector, truncating the

matrix deletes only the least relevant data. The finished matrix is not necessarily more dense, but it contains more relevant ratings.

| Items \ Users | U1 | U2 | U3 | U4 | U5 |
|---|---|---|---|---|---|
| I1 | | 1 | 2 | 2 | 3 |
| I2 | 3 | 4 | 3 | 4 | |
| I3 | 1 | | 3 | 2 | 2 |
| I4 | 4 | 3 | | 4 | 2 |
| I5 | 4 | | 3 | 5 | |

Figure 5.3: Input Matrix Generation – Load Matrix

| Items \ Users | U1 | U2 | U3 | U4 | U5 |
|---|---|---|---|---|---|
| I1 | | 1 | 2 | 2 | 3 |
| I3 | 1 | | 3 | 2 | 2 |
| I4 | 4 | 3 | | 4 | 2 |
| I5 | 4 | | 3 | 5 | |
| I2 | 3 | 4 | 3 | 4 | |

Figure 5.4: Input Matrix Generation – Sorting by Items/Rows

Users / Items

| Items | U1 | U4 | U2 | U3 | U5 |
|---|---|---|---|---|---|
| I1 | | 2 | 1 | 2 | 3 |
| I3 | 1 | 2 | | 3 | 2 |
| I4 | 4 | 4 | 3 | | 2 |
| I5 | 4 | 5 | | 3 | |
| I2 | 3 | 4 | 4 | 3 | |

Figure 5.5: Input Matrix Generation – Sorting by Users/Columns

Users / Items

| Items | U1 | U4 | U2 |
|---|---|---|---|
| I1 | | 2 | 1 |
| I3 | 1 | 2 | |
| I4 | 4 | 4 | 3 |

| U3 | U5 |
|---|---|
| 2 | 3 |
| 3 | 2 |
| | 2 |
| 3 | |
| 3 | |

| | | | | | |
|---|---|---|---|---|---|
| I5 | 4 | 5 | | 3 | |
| I2 | 3 | 4 | 4 | 3 | |

Figure 5.6: Input Matrix Generation – Truncate Matrix

## 5.2 Two Ways of Establishing Neighbor Influence

A major assumption in many collaborative filtering approaches is that similar users agree on a majority of items, regardless of their domain. We refer to this type of comparison as "global similarity" because ratings from every domain contribute to the similarity of any two users. This concept establishes strong connections among neighbors, but it eliminates potentially good vectors because they are not similar enough. On the other hand, a neighbor could be chosen because he/she shares many similar ratings in unrelated domains, yet offers little useful information in the active item domain.

It is natural for people to agree on some items and disagree on others. In fact, Bell and Koren suggest that the reason so many recommender systems fail is because they assume that the similarity between any two users is fixed across all items [13]. This is unlikely, because each user has a unique set of preferences. For instance, it is possible for two users to be considered similar in horror and drama genres, yet have nothing in common in their comedy preferences. Therefore, we should compute user similarity only in the context of the active item.

Our proposed solution satisfies this need as it only considers users to be similar if they agree on a subset of all items. In other words, our approach sorts the columns of a matrix according to the most relevant rows. We refer to this type of comparison as "local similarity" because it considers the information from only a few locally relevant domains. We loosen user similarity requirements and allow for more potentially good neighbors. Neighbors are also more relevant, since they explicitly demonstrate their similarity in the active item domain.

The large size and extreme dataset sparsity is the reason global similarity is inaccurate. In fact, existing research compared standard, whole-matrix recommendations to three local similarity approaches to discover that accurate suggestions may be made with a small, but relevant, fraction of the available data [6, 21]. One of the approaches relies on a heuristic assumption that similar users remain similar across all user models, i.e., movie genres. We claim that this assumption is false because there are far less people who agree on everything than there are people who share some preferences. The authors support our claim and show that this particular approach does not provide accurate recommendations.

The local similarity approach offers more flexibility over the global similarity method. Other publications also advocate using local similarity as opposed to the traditional approach [38, 130]. However, there is no automatic way to identify the most relevant items by which the users should be compared. One way to do so is to partition the dataset according to items and to consider all items in the same partition to be relevant to each other [35]. This way, when making a suggestion about an item, all users are compared according to items in the same partition. For example, Berkovsky, Kuflik, and Ricci partition the dataset according to eight movie genres and consider each partition to be a distinct user model [19, 21, 22]. The partitions are essentially smaller user-item matrices containing all user vectors and a subset of item vectors. Such a composite model of a user is accurate, especially if partitions are allowed to overlap to represent mixed genre movies.

The manual partition approach is similar to the one we are proposing. Both methods rely on local similarity to compare vectors on the dimensions that matter the most.

The manual partitioning approach uses items within the same partition to compare users. We use more explicit item similarities to determine the most important dimensions. Instead of using static domain boundaries, our approach dynamically creates a single partition around the active rating. We form our partition across items as well as users. As a result, our algorithm generates an input matrix with a carefully selected set of ratings.

Another way to discriminate dimensions is to weigh them. The weighted approach considers some properties of the vector to be more important than others [106]. For example, when comparing two user vectors, their opinions of certain items matter more than others. The items that are most similar to the active item should receive the most weight. Therefore, we use item similarities as dimension weights for computing user similarities and we use user similarities as dimension weights for computing item similarities.

## 5.3 Candidate Input Generation Algorithms

Our empirical study considers three different input selection methods and two similarity measures. Some approaches use raw cosine similarity to sort the matrix:

$$sim(a, u) = \frac{\sum_i (r_{a,i} * r_{u,i})}{\sqrt{\sum_i (r_{a,i})^2}\sqrt{\sum_i (r_{u,i})^2}}$$

Other approaches compute the similarity among vectors as a Pearson's correlation. It is similar to cosine similarity, except each vector property is normalized by the mean of that vector:

$$sim(a, u) = \frac{\sum_i (r_{a,i} - \bar{r}_a)(r_{u,i} - \bar{r}_u)}{\sqrt{\sum_i (r_{a,i} - \bar{r}_a)^2}\sqrt{\sum_i (r_{u,i} - \bar{r}_u)^2}}$$

Note that if $i$ goes through every property of the active vector, we refer to this measure as global similarity. Otherwise, if $i$ goes through some of the properties, we refer to this measure as local similarity.

The standard approach sorts the matrix according to either of the similarity metrics and truncates it down. It uses global similarity to rank matrix dimensions. It is similar to other methods of selecting input with one notable exception, bidirectional truncation. Existing approaches compute vector similarity and truncate the matrix across only one dimension, i.e., either users or items [14, 56, 107]. We truncate both dimensions. Our approach is slower, but it allows the same input to be used by user and item-oriented algorithms.

The weighted approach considers some properties of the vector to be more important than others. We use item similarities as weights for computing user similarities and user similarities as weights for computing item similarities. The weighted approach users existing global similarities as weights [7]. Furthermore, we can use the cosine similarity or Pearson's correlation measure to compare vectors with weighted dimensions. We consider all four possibilities in our case study.

The recursive approach recursively selects the top 30 vectors among users and items. On the first iteration we compare vectors on all shared properties, i.e., global similarity. However, with each additional iteration the similarity within one dimension is based exclusively on the top vectors in the other dimension, i.e., local similarity. For instance, user similarity is determined only along the top item opinions. Likewise, item similarity is computed only from the top users' opinions. We make a new list of the top 30 vectors with each iteration. The vectors may be chosen based on either

similarity metric. We consider five different configurations of this approach over two, three, and four iterations.

## 5.4 Desired Input Qualities

In this section we make a few observations regarding an algorithm input that usually results in an accurate recommendation. We derive some observations from the underlying assumptions of the KNN approach. Some observations are based on empirical data from previous research and our own experiments. However, all these observations are necessary to support the theory behind our recursive input generation algorithm.

### 5.4.1 Bigger Net Weights Produce More Reliable Results

The concept of collaborative filtering is built on the idea that users who agreed in the past are likely to agree in the future. As a result, users who agree more tend to have a bigger influence. Ideally, the opinions are unanimous and every neighbor has a weight of 1. In the worst case scenario, everyone's weight is 0, which means that neighbors have no similar opinions. In reality, the weights are somewhere in the middle, with no irrelevant vectors, because at least one shared opinion is required to be considered a neighbor.

Consider a neighborhood of $n$ vectors, where each one has a weight $w_i \in (0, 1]$. The net weight of the neighborhood will be $\sum_{i=1}^{n} w_i \in (0, n]$. Therefore, a good input generation algorithm should aim to produce a neighborhood with a maximum net weight, because that is closer to the best case scenario. One way to do that

is to discard the vectors with the lowest weights. Such vectors do not contribute enough weight to justify the additional noise they produce. Therefore, it is sometimes beneficial to ignore certain neighbors, even if their vectors increase the net weight of a matrix.

Low weights reduce recommendation accuracy. To prove this, consider a user-oriented KNN recommendation that computes the expected value of a rating $r$ by a user $a$ on an item $i$, $E[r_{a,i}]$. The estimate, $P_{a,i}$, is an average of neighbors' ratings on the active item, weighted by their respective similarities:

$$P_{a,i} = E[r_{a,i}] = \frac{\sum_{u \in U_i} r_{u,i} w(a,u)}{\sum_{u \in U_i} w(a,u)}$$

Note that this formula does not contain the $k$ parameter, i.e., $k = 1$. It adjusts overly optimistic/pessimistic estimates and has no effect on the averaging procedure. The simplified formula also uses raw ratings instead of normalized values. Normalization reduces the scale of the estimation, with no effect on the recommendation process.

The Mean Squared Error (MSE) of such recommendation is proportional to the variance of the estimate:

$$MSE = E(P_{a,i} - r_{a,i})^2 = Var(P_{a,i}) + Bias(P_{a,i}, r_{a,i})^2$$

We assume that bias term is zero or constant, because the KNN algorithm always produces the same recommendation from the same neighborhood. In other words, this approach cannot distinguish between two vectors with similarity of 1. Furthermore, a constant bias may be neutralized with the $k$ term. Therefore, prediction variance

is a good indicator of recommendation accuracy, i.e., MSE:

$$MSE = var(P_{a,i}) = E(P_{a,i}^2) - E(P_{a,i})^2$$

Consider a recommendation based on two neighbors and its variance:

$$P_{a,i} = \frac{r_1 w_1 + r_2 w_2}{w_1 + w_2}$$

$$var(P_{a,i}) = \frac{r_1^2 w_1 + r_2^2 w_2}{w_1 + w_2} - \left(\frac{r_1 w_1 + r_2 w_2}{w_1 + w_2}\right)^2$$

After simplifying this equation we get the following:

$$var(P_{a,i}) = \frac{r_1^2 w_1 + r_2^2 w_2}{w_1 + w_2} - \frac{r_1^2 w_1^2 + 2 r_1 r_2 w_1 w_2 + r_2^2 w_2^2}{(w_1 + w_2)^2} = \frac{w_1 w_2 (r_1 - r_2)^2}{(w_1 + w_2)^2}$$

Taking a partial derivative of the estimation variance with respect to one of the weights gives us an idea of how the individual weights affect recommendation accuracy:

$$\frac{d}{dw_1} var(P_{a,i}) = (r_1 - r_2)^2 \left[\frac{w_2}{(w_1 + w_2)^2} - \frac{2 w_1 w_2}{(w_1 + w_2)^3}\right] = \frac{(r_1 - r_2)^2}{(w_1 + w_2)^3}\left[-w_1 w_2 - w_2^2\right] < 0$$

The partial derivative with respect to any one weight is going to be negative for any rating scale, as long as the weights are positive. In other words, the variance will increase as individual vector similarities become smaller. Likewise, higher weights produce smaller variance and result in a more accurate recommendation, i.e., smaller MSE. By induction, the same principle applies to cases with any number of weights. In other words, we can regroup a set of $n$ weights into two new weights, $w_1'$ and $w_2'$:

$$P_{a,i} = \frac{\sum_{u=1}^{n} r_{u,i} w(a,u)}{\sum_{u=1}^{n} w(a,u)} = \frac{\sum_{u=1}^{n-1} r_{u,i} w(a,u) + r_{n,i} w(a,n)}{\sum_{u=1}^{n-1} w(a,u) + w(a,n)} = \frac{r_1' w_1' + r_2' w_2'}{w_1' + w_2'}$$

Taking a partial derivative with respect to the last weight would also produce a negative result, thus proving that smaller weights harm recommendation accuracy.

### 5.4.2 Sorting Before Truncating for Maximum Net Weight

A set of vectors ordered by their influence will always produce a net weight greater than that of a random sample of neighbors. Consider two cases: a list of sorted weights $S = w'_1, w'_2, w'_3, ..., w'_n$ such that $w'_1 >= w'_2 >= w'_3 ... >= w'_{n-1} >= w'_n$ and the same set of weights in no particular order $R = w_1, w_2, w_3, ..., w_n$. Let us choose the first $m$ elements from both sets and order them in descending order. Then, $\sum_{i=1}^{m} w_i <= \sum_{i=1}^{m} w'_i$ because $w'_i >= w_i$ for all $i$. Sorting ensures that after truncation the neighborhood contains most similar vectors as opposed to a random sample of them. Without sorting, a matrix may still have a high net weight, but such an outcome is unlikely.

### 5.4.3 Global Similarity does not Imply Local Similarity

The standard approach incorrectly assumes that high global similarity guarantees high local similarity in every domain. For instance, two globally similar vectors may disagree on a few dimensions. As long as the number of such dimensions is sufficiently small, the global similarity remains high. However, local similarity according to these dimensions would conclude them to be less similar. Likewise, one can choose dimensions from globally dissimilar vectors and get high local similarity.

Consider a case with two vectors of three dimensions, $\vec{a} =< 1, 2, 1 >$ and $\vec{b} =< 1, 2, 5 >$. The global cosine similarity between these two vectors compares them on

all three dimensions:

$$sim(a, b) = \frac{1 * 1 + 2 * 2 + 1 * 5}{\sqrt{1^2 + 2^2 + 1^2}\sqrt{1^2 + 2^2 + 5^2}} = \frac{10}{\sqrt{6}\sqrt{30}} = 0.745$$

However, the local cosine similarity between the same two vectors may be lower or higher, depending on which dimensions are used. The local similarity between the two vectors across the first and second dimension is higher than their global similarity:

$$sim(a, b) = \frac{1 * 1 + 2 * 2}{\sqrt{1^2 + 2^2}\sqrt{1^2 + 2^2}} = \frac{5}{\sqrt{5}\sqrt{5}} = 1$$

However, the local cosine similarity between the two vectors across the first and third dimension is lower than their global similarity:

$$sim(a, b) = \frac{1 * 1 + 1 * 5}{\sqrt{1^2 + 1^2}\sqrt{1^2 + 5^2}} = \frac{6}{\sqrt{2}\sqrt{26}} = 0.48$$

A neighborhood with a great global similarity net weight does not necessarily have a great local similarity net weight.

This distinction is important because the KNN algorithm weighs neighbors' opinions according to the data in the input, not the original dataset. The input matrix contains just the ratings, so vector similarities must be recomputed. In other words, the global similarities are lost between the input generation and combination algorithms. Because the input matrix is a truncated version of the original dataset, any similarities that are computed from it are based on a subset of all shared dimensions. Therefore, local similarity determines a neighbor's influence, but since high global similarity does not guarantee high local similarity, the net weight of an input matrix is not as high as it could be.

The standard approach produces low local similarity net weights because the neighbors are not explicitly selected to exhibit high local similarity according to a

subset of shared dimensions. As a result, the input matrix contains a random sample of local similarities. Since these weights were not chosen in order of decreasing value, their sum is not guaranteed to be maximum. In other words, a matrix that has been sorted only once will not have the maximum local similarity net weight. In order to guarantee that only the most influential neighbors are considered, the matrix needs to be resorted. Our recursive approach recomputes local similarity and rearranges the vectors such that the net weight of the input matrix is maximized. The resulting neighborhood is closer to the ideal scenario, so the resulting recommendations should be more trustworthy.

### 5.4.4 Second Pass Local Similarities are Higher

Since global and local similarities are unrelated, it is possible that comparing two vectors on fewer dimensions could produce a higher similarity. Consider a case where we sort the matrix by rows and columns such that the most similar vectors are positioned closer to the top left corner of the matrix. In this example, the sum of row Pearson's correlations decreases with $n$, if the similarity of columns decreases:

$$\sum \frac{\sum_i^n (r_{a,i} - \bar{r}_a)(r_{u,i} - \bar{r}_u)}{\sqrt{\sum_i^n (r_{a,i} - \bar{r}_a)^2}\sqrt{\sum_i^n (r_{u,i} - \bar{r}_u)^2}} >= \sum \frac{\sum_i^{n+1} (r_{a,i} - \bar{r}_a)(r_{u,i} - \bar{r}_u)}{\sqrt{\sum_i^{n+1} (r_{a,i} - \bar{r}_a)^2}\sqrt{\sum_i^{n+1} (r_{u,i} - \bar{r}_u)^2}}$$

Assuming that dimensions are sorted, considering less similar dimensions will cause $r_{u,i} - \bar{r}_u$ to grow, thus allowing outliers on a neighbor's rating scale to affect his/hers similarity. Extreme opinions on different scales are less likely to agree, so the similarity of such vectors would decrease. If this is false, then considering an extra dimension will result in higher similarity. If that were the case, the additional dimension should

be considered more similar than the first $n$, since all rows tend to agree on it. However, this is impossible because the dimensions are considered in order of decreasing similarity.

Consider two vectors with two dimensions each, $\vec{a} =< a_1, a_2 >$ and $\vec{b} =< b_1, b_2 >$. Assuming that their dimensions have been sorted according to their cosine similarity with the first column, the following inequality is true:

$$\frac{a_1 a_2 + b_1 b_2}{\sqrt{a_1^2 + b_1^2}\sqrt{a_2^2 + b_2^2}} \leq \frac{a_1 a_1 + b_1 b_1}{\sqrt{a_1^2 + b_1^2}\sqrt{a_1^2 + b_1^2}}$$

In other words, the first column's similarity to itself is greater than its similarity with the second column. By definition, when a vector is compared to itself, its cosine similarity should be 1. After simplification, we derive the following inequality:

$$(a_1 a_2 + b_1 b_2)^2 \leq (a_1^2 + b_1^2)(a_2^2 + b_2^2)$$

$$2 a_1 a_2 b_1 b_2 \leq a_2^2 b_1^2 + a_1^2 b_2^2$$

Such inequality works for any vector $a$ and $b$:

$$0 \leq a_2^2 b_1^2 - 2 a_1 a_2 b_1 b_2 + a_1^2 b_2^2 \leq (a_2 b_1 - a_1 b_2)^2$$

Taking a square root of both sides, we get:

$$0 \leq \pm(a_2 b_1 - a_1 b_2)$$

$$a_1 b_2 \leq a_2 b_1$$

$$a_1 b_2 \geq a_2 b_1$$

The above two inequalities cover the entire number range. Therefore, the original assumption holds true for any value of $a_1, a_2, b_1, b_2$.

Given sorted dimensions, we can prove that considering additional dimensions reduces $\vec{b}$ vector similarity:

$$\frac{a_1 b_1 + a_2 b_2}{\sqrt{a_1^2 + a_2^2}\sqrt{b_1^2 + b_2^2}} \leq \frac{a_1 b_1}{\sqrt{a_1^2}\sqrt{b_1^2}}$$

In other words, the similarity between vectors $a$ and $b$ will be greater when we compare them on a single dimension than if we compare them on two dimensions, provided that dimensions are considered in order of decreasing similarity:

$$\frac{a_1^2 b_1^2 + 2a_1 a_2 b_1 b_2 + a_2^2 b_2^2}{a_1^2 b_1^2 + a_2^2 b_1^2 + a_1^2 b_2^2 + a_2^2 b_2^2} \leq 1$$

$$2a_1 a_2 b_1 b_2 \leq a_2^2 b_1^2 + a_1^2 b_2^2$$

Furthermore, according to the previous proof, this inequality holds true for any two vectors. Likewise, comparing higher-dimension vectors would produce similar results. The individual vector weights may change differently, but their net sum will decrease as we consider more dimensions with lower similarities.

## 5.5 Similarity Refinement Simulation

To simulate the recursive input generation algorithm, consider five vectors with five dimensions $< a, b, c, d, e >$ in Figure 5.7. The net weight of such matrix is 3.56, where the weight of each row is computed as its Pearson's correlation to the first row. The net weights of four truncated versions of this matrix, with one of the dimensions removed, are as follows: no $e = 3.87$, no $d = 4.08$, no $c = 3.38$, no $b = 2.94$. Clearly, considering all dimensions except $d$ produces the biggest net weight, but how does one know which dimensions reduce the net weight?

| a | b | c | d | e |
|---|---|---|---|---|
| 3 | 4 | 2 | 1 | 3 |
| 2 | 3 | 1 | 1 | 3 |
| 3 | 4 | 3 | 3 | 4 |
| 4 | 4 | 2 | 2 | 4 |
| 3 | 3 | 2 | 3 | 2 |

Figure 5.7: A Matrix with Five Vectors and Five Dimensions

Let us consider the similarity of each dimension to $a$: a = 1.00, b = 0.65, c = 0.50, d = 0.35, e = 0.42. The $d$ and $e$ dimensions have the smallest correlations and removing them increases the net weight. In fact, there is a negative correlation between the similarity of a dimension and the net weight of a truncated matrix that does not contain it. Figure 5.8 shows this relationship.

To verify this phenomenon, we examined the local weights of input matrices generated by the two algorithms on the Netflix dataset. Figures 5.9 and 5.10 show typical similarities of the first 30 vectors in an item and user-oriented matrix. The first vector is always the active vector, so its similarity to itself is always 1. However, as we inspect the similarity of the neighbors, the vectors selected by the recursive algorithm are consistently more similar. Both graphs demonstrate that the recursive approach generates higher similarities with greater net weights, which produce more confident recommendations.

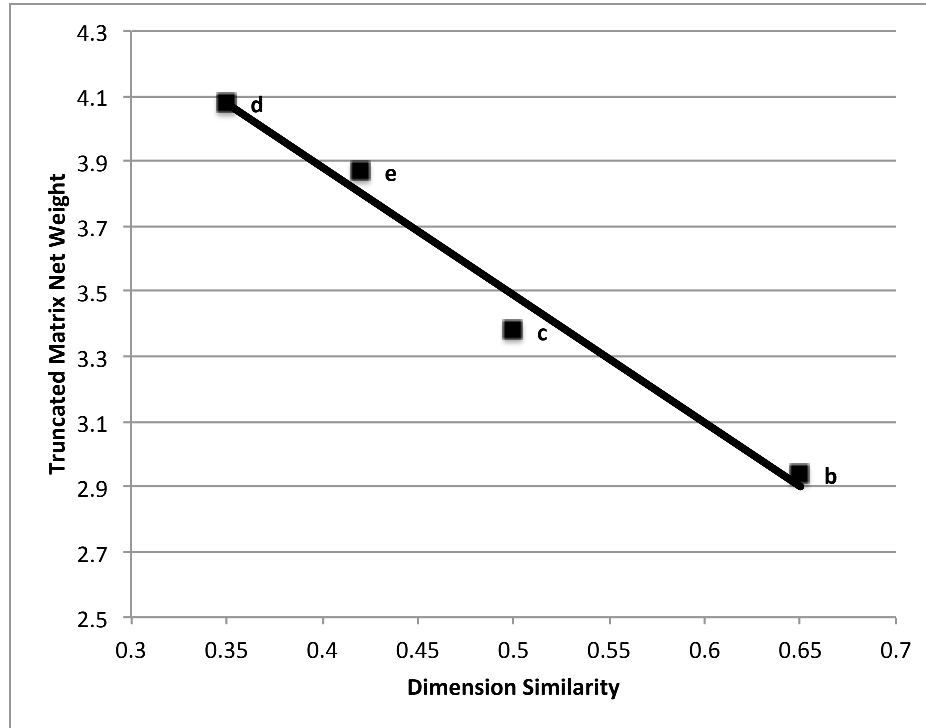


Figure 5.8: Truncated Matrices and Their Net Weights

## 5.6 The Effects of Input Resorting on Estimation Accuracy

To further support the benefits of our recursive algorithm, we considered a statistical justification of this approach. The Rao-Blackwell theorem states that if $g(x)$ is an estimator for $\theta$, then conditional expectation of $g(x)$ given a sufficient statistic $T(x)$ is a better estimator of $\theta$ and never worse [25]. At its core level, this theorem employs a well-known relationship between conditional and unconditional variance, i.e., $var(E(g(x) \mid T(x))) <= var(E(g(x)))$ [32]. In practice, averaging over a sufficient statistic does not lead to an increase of the mean squared error [32], which represents the accuracy of a recommender system. Therefore, in order to improve the accuracy,

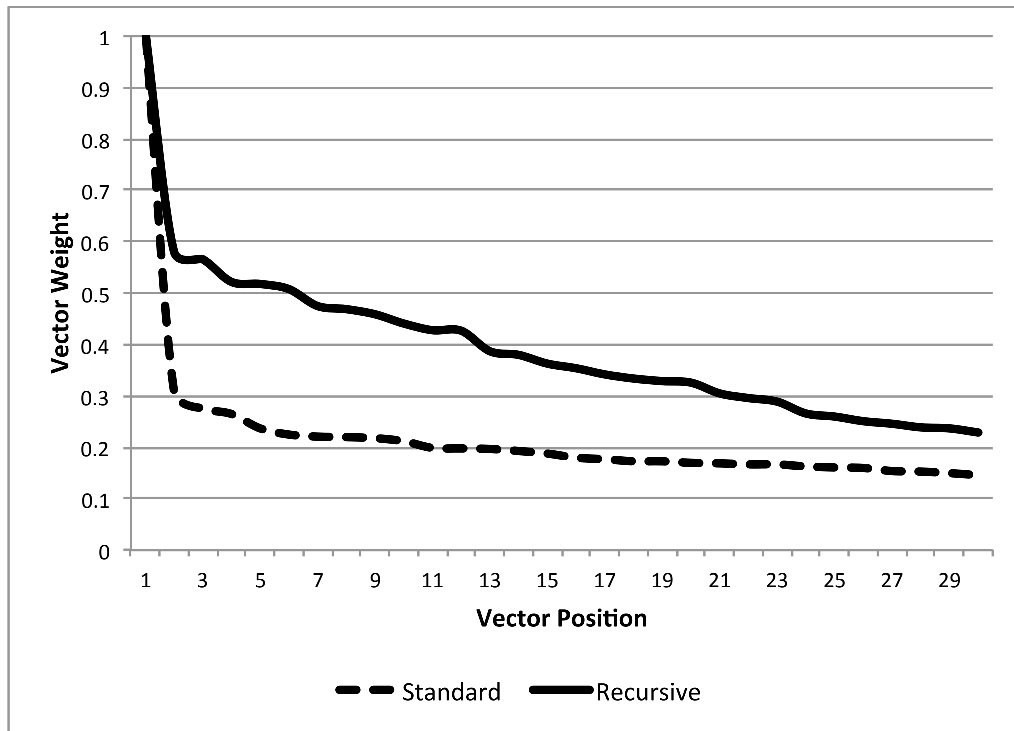


Figure 5.9: Typical Similarities of the First 30 Vectors in an Item-Oriented Matrix

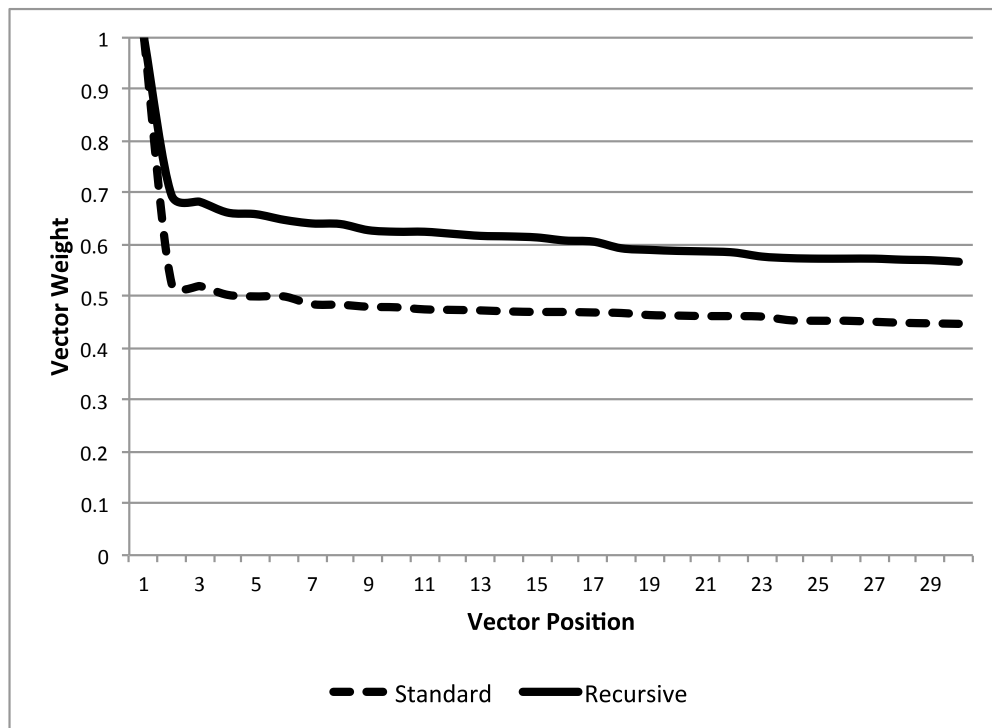


Figure 5.10: Typical Similarities of the First 30 Vectors in a User-Oriented Matrix

we should look for estimators which are functions of the sufficient statistic.

A sufficient statistic is a function of data that describes it in such a way that a sample generated according to this statistic would be as useful as a data sample for estimating $\theta$ [78]. In the context of our input generation algorithm, $\theta$ is the actual rating we are trying to predict and the initial vector similarities represent the sufficient statistic. The purpose of the sufficient statistic is to capture all of the useful information necessary for estimating $\theta$, so that the data may be discarded in favor of the statistic. Our recursive algorithm computes vector similarities on the first pass and discards the data because we already have all the useful information about it, i.e., which dimensions are best for predicting this active rating.

The purpose of statistical sufficiency is to ensure that the resulting conditional expectation is a reliable estimator. If the statistic is not sufficient, the result could depend on some unknown parameters, which would defeat the purpose of this approach [117]. A nice property of the sufficient statistic is that its expected value is equal to $\theta$, yet the statistic may not depend on $\theta$. The first pass of the recursive algorithm is a sufficient statistic for the relevant ratings dataset. Global vector similarities provide all the information necessary for estimating the active rating. However, they do not depend on the active rating because it is possible to make two input matrices with the same global similarities, but different active ratings.

Our algorithm uses global similarities to partition the relevant dataset. It essentially categorizes the matrix dimensions into two groups, the top 30 and everybody else. Vectors from the first group receive a weight of $T(x) = 1$ and everyone else receives a weight of $T(x) = 0$. In other words, the first pass of the recursive algorithm

decides which dimensions are the most relevant. Doing so improves the accuracy of an existing estimator $g(x)$, i.e., a weighted average of known ratings and local similarity weights.

The existing estimator does not have to be perfect. As we have demonstrated in section 5.4.3, computing local similarity often leads to different results, depending on the choice of matrix dimensions. However, combining local similarity with global similarity partitioning, or Rao-Blackwellisation of $g(x)$, produces a more optimal estimator.

Relying on either similarity approach individually will not produce accurate estimates, but their combination will. The local similarity weights do not carry all the information that is useful for estimating $\theta$, but this information is available in the relevant dataset and is therefore captured by $T(x)$. The global similarity does have all the information for estimating $\theta$, but it is not an estimator [109]. The conditional expected value of a rough estimator given a sufficient statistic $E(g(x) \mid T(x))$ is an average of all ratings that have the same value for $T(x)$. This combination reduces sensitivity to any particular rating and utilizes all useful information in the dataset because $T(x)$ is a sufficient statistic and its value is fixed throughout the averaging procedure [48]. Our approach takes advantage of this statistical property by computing local similarity weights according to the dimensions selected by their global similarity ranking.

## 5.7 Conclusion

This chapter describes our process for identifying relevant ratings. One of the key concepts of this approach is the requirement for two neighbors to be similar in some, but not all domains. We observe this quality in our simulation as well as best-case scenario considerations for the collaborative filtering approach. Therefore, we incorporate local similarity measure into our input generation algorithm. In order to generate a high quality input, we first organize all ratings in a matrix and identify a few valuable dimensions. We then reorder the matrix according to these dimensions and truncate it to size. Establishing the important dimensions first reduces random variability within the data, which gives us a better idea of true user preferences. As a result, our algorithm can reduce the size and increase the quality of recommender input.

# Chapter 6

# A Novel Input Generation Model

Previous chapters presented a theory that explains how our approach could improve recommendation accuracy. They identified the overall structure of our system, its components, and connections. Such a high-level view of our system raised the level of abstraction at which we reasoned about the desired functionality. With no implementation details to worry about, we focused on other important system properties like scalability, performance, and adaptability. To achieve them, we used well-known architecture styles that have recognizable benefits with established implementations.

The architecture we presented earlier is informal for a reason. It is a simple and intuitive way to characterize our system, without imposing unnecessary restrictions on how it should work. However, the informal model provides little information about the actual computations represented by boxes, their interfaces, or the nature of interactions between them [4]. The lack of such details means that our input generation component may be implemented in many different ways. In order to simplify and encourage future development around our input generation component,

we formalize its design.

## 6.1 Formalism Motivation, Scope, and Goals

In order for the input generation component to achieve its goals, its interfaces must be consistent, the functions within it must agree on the order of execution, and the purpose of each function should be explicit [151]. To accomplish this, we develop a collection of Z schemas that comprise the semantic model of the input generation component. These simple, compact, and informative models use standard mathematical notation to document the design of the most important part of our system. They also form a solid blueprint for future system construction, deployment, and execution.

A formal notation communicates our design decisions without committing to a particular implementation. The Z schemas highlight each function's contributions to the overall component functionality and provide a unique perspective on how the algorithm works. They suppress implementation details, so we can concentrate on the analysis and decisions that are most crucial to satisfying the component's requirements [4]. They also make implementation considerably easier because our decisions and rationale are explicitly documented.

A typical Z specification, called schema, is a named predicate that constrains some aspect of the application. It consists of two parts that describe the structure and behavior of the model. The static schemas provide a fixed view of the model [47] and refine our existing entity-relationship diagrams. The behavior schemas show how the state of the model changes [47] and refine the existing procedural descriptions.

Each change is described in terms of preconditions existing before that change and postconditions that must be true after the change. The following sections describe our input generation component in terms of structure and behavior specifications.

### 6.1.1 Static Model Structure

First, we establish the types of variables in our system. We identify every user and item with a positive integer. This is a compact and convenient way to refer to both kinds of vectors. Additionally, the generic naming convention allows us to represent various datasets regardless of their domain. Furthermore, numeric IDs are not related to other vector properties, which makes them ideal unique identifiers. The following is a formal definition of the ID and rating types:

$$ID : \mathbb{N}$$
$$RATING : \mathbb{N}$$

---

$$ID > 0$$
$$RATING = 0..5$$

We require that opinions are expressed as numerical ratings. For this particular dataset, the ratings must be on a scale from one to five. We reserve the value of zero for missing ratings that must participate in the similarity computation. However, the scale may change independently of the rest of the model. In other words, our model will perform equally well on a dataset with discrete or continuous ratings.

The following definition builds on the previous one and introduces two new concepts. Vector is a collection of ratings identified by dimension IDs. For instance, a

user vector would be a collection of item ratings. Each rating in such a vector is related to a particular item ID. Likewise, an item vector is a collection of ratings identified by user IDs. The sim function compares two vectors by quantifying the difference among related ratings on a continuous scale from -1 to 1. This restriction admits cosine similarity range of [0..1], because all ratings are positive, whole numbers. It also allows Pearson's correlation with the range of [-1..1], because normalized ratings may be negative. We define the vector data type and the sim function as follows:

$$VECTOR : ID \nrightarrow RATING$$
$$SIM : VECTOR \times VECTOR \rightarrow \mathbb{R}$$

$$\operatorname{ran} SIM = -1..1$$
$$a, b : VECTOR \mid a \cap b = \emptyset \Leftrightarrow SIM(a, b) = 0$$
$$a : VECTOR \mid SIM(a, a) = max\,\{\operatorname{ran} SIM\}$$

Sim is a total function, which means that it should be able to compute the similarity between any two vectors. If the intersection between two vectors is empty, the similarity must be zero. Likewise, if the similarity between two vectors is zero, they have no dimensions in common. Also, when a vector is compared to itself, its similarity must be maximum. This ensures that sorting does not discard the active vectors, which form a point of reference for further similarity computations.

We plan to store the dataset in a relational database, which is a collection of relations. Each relation has a rigid structure, which may be viewed as a set of records. Each record may be decomposed into a set of simple atomic values [47]. Previous schemas established the types of data we work with. The *record* schema

uses these data types to establish the structure of our dataset:

*RECORD*

$user_id : ID$

$item_id : ID$

$rating : RATING$

The following schema describes the original read-only dataset. It is a set of records that associate a user-item tuple with a particular rating. Each tuple is unique within this dataset, so there should be as many unique tuples as there are ratings. A sparse dataset does not contain a rating for every possible user-item combination, yet each user and item vector must contain at least one rating. Likewise, every rating in the dataset must belong to a known user and item. The *dataset* schema captures all of these constraints:

*dataset*

$users : \mathbb{P}\, ID$

$items : \mathbb{P}\, ID$

$ratings : \mathbb{P}\, RECORD$

$A : ratings \bullet \#(A.user_id, A.item_id) = \#ratings$

$u : ID, A : ratings \mid \forall\, u \in users \bullet \#\{A.user_id = u\} \geq 1$

$i : ID, A : ratings \mid \forall\, i \in items \bullet \#\{A.item_id = i\} \geq 1$

$r : RECORD \mid \forall\, r \in ratings \bullet \{r.user_id \in users\} \wedge \{r.item_id \in items\}$

To ensure reliable experiment results, the input generation algorithm should never modify the dataset. This way, we can compare the accuracy of different approaches under the same conditions. Also, the dataset should never be empty. In fact, it

should represent at least two user and item vectors. Even though it is possible to make generic recommendations from as little as two vectors and a single rating, such dataset does not contain enough information to establish vector similarity. Therefore, we require more data to establish personalized recommendations. Figures 6.1 and 6.2 show the minimum amount of data necessary for our approach. These constraints are also encoded in the following schema:

$\Xi dataset$

$dataset$

---

$users' = users$

$items' = items$

$ratings' = ratings$

$\#users \geq 2$

$\#items \geq 2$

$\#ratings \geq 3$

Our model requires the dataset to contain at least two user vectors, two item vectors, and three ratings. Such dataset requirements enable the similarity measure, so we can reason about the relevancy of the available data. In other words, the similarity measure will be non-zero for some vectors because they are bound to have common dimensions. For example, the cosine similarity measures the cosine of the angle between two vectors. Therefore, there must be at least two vectors to compare. Furthermore, each vector must have more than one dimension, because the angle between any two scalar values is always zero.

The *matrix* schema describes the model of the actual recommender input. It is

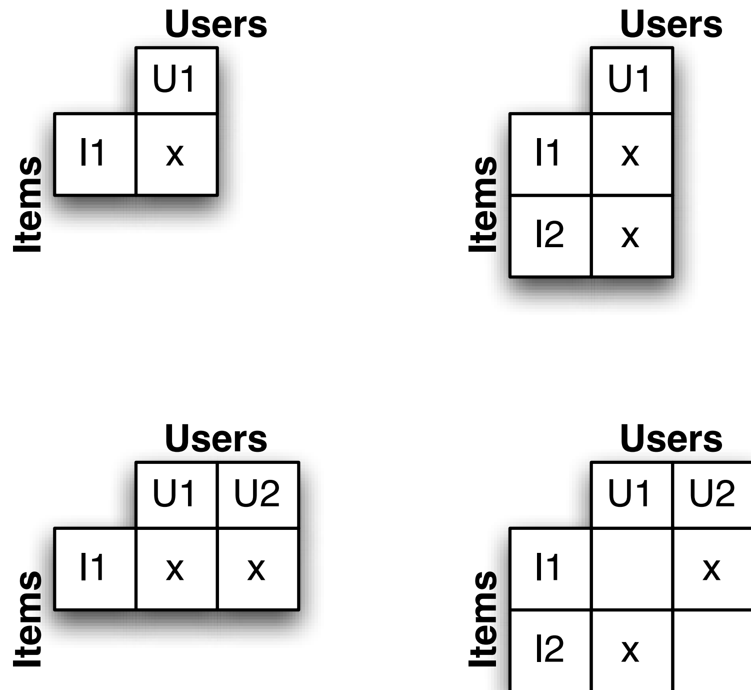


Figure 6.1: Examples of Inappropriate Minimum Datasets

similar to the original dataset in a sense that it keeps track of the users, items, and their ratings. However, unlike the original read-only dataset, matrices may shrink. In other words, once the matrix has been initialized, the data within it may be removed. The matrix also keeps track of the best dimensions, which may not be empty. As the name implies, best users form a subset of all relevant users and best items form a subset of all relevant items. Note that best dimensions contain a sequence of IDs, because the order in which they are removed makes a difference. The matrix structure is defined in the following schema:

| Items \ Users | U1 | U2 |
|---|---|---|
| I1 | x | x |
| I2 | x | |

Figure 6.2: An Appropriate Minimum Dataset

*matrix*

$$users : \mathbb{P}\, ID$$

$$items : \mathbb{P}\, ID$$

$$ratings : \mathbb{P}\, RECORD$$

$$best_users : \mathrm{seq}\, ID$$

$$best_items : \mathrm{seq}\, ID$$

---

$$A : ratings \bullet \#(A.user_id, A.item_id) = \#ratings$$

$$u : ID, A : ratings \mid \forall\, u \in users \bullet \#\{A.user_id = u\} \geq 1$$

$$i : ID, A : ratings \mid \forall\, i \in items \bullet \#\{A.item_id = i\} \geq 1$$

$$r : RECORD \mid \forall\, r \in ratings \bullet \{r.user_id \in users\} \wedge \{r.item_id \in items\}$$

$$best_users \neq \emptyset$$

$$best_items \neq \emptyset$$

$$best_users \subseteq users$$

$$best_items \subseteq items$$

### 6.1.2 Data Access Behavior Specification

When working with little data, the generated input would essentially contain the entire dataset. However, there are cases when some data is purposefully omitted. The following schema describes the process of selecting relevant ratings. According to the previously established architecture, the request must contain two IDs, which represent the active user ID and active item ID. The output contains a set of users with a rating on the active item and a set of items with a rating by the active user, i.e., matrix dimensions. The algorithm should also return all ratings associated with these matrix dimensions. The following schema specifies the behavior of the first step of the matrix generation process:

*get_matrix*

$\Xi dataset$

$active_user_id? : ID$

$active_item_id? : ID$

$users! : \mathbb{P}\, ID$

$items! : \mathbb{P}\, ID$

$ratings! : \mathbb{P}\, RECORD$

---

$$users! = \{A : ratings \mid A.item_id = active_item_id? \bullet A.user_id\} \cup active_user_id?$$

$$items! = \{A : ratings \mid A.user_id = active_user_id? \bullet A.item_id\} \cup active_item_id?$$

$$ratings! = \{A : ratings \mid A.user_id \in users! \wedge A.item_id \in items!\} \oplus \{active_user_id?, active_item_id?, 0\}$$

The matrix should always contain the active vectors, even if the active rating is unknown. To ensure this, we include the active user ID in the users list and

active item ID in the items list. To prevent a known active rating from affecting the recommendation process that attempts to guess it, we add an extra relation to the ratings set that overwrites the known rating, if there is one. Note that a value of zero is reserved for a missing rating. Therefore, we are forcing the system to pretend like the active rating does not exist.

The following schema describes the initial state of the matrix. It accepts a set of users, items, and rating records generated from the original dataset. Note that the best dimensions may not be empty, even in the initial version of the matrix. Therefore, we initialize the best users and best items lists with every available dimension. This way, the best dimensions satisfy all matrix requirements:

$\Delta init_matrix$

$matrix$

$users? : \mathbb{P}\, ID$

$items? : \mathbb{P}\, ID$

$ratings? : \mathbb{P}\, RECORD$

---

$users' = users?$

$items' = items?$

$ratings' = ratings?$

$best_users' = users?$

$best_items' = items?$

### 6.1.3 Data Processing Behavior Specification

The matrix may change in two ways, sorting or truncation. The purpose of sorting is to compute the new best dimensions. The sorting process uses the data in the

matrix to do that, but that data should remain unchanged. This way, any subsequent sorting procedures will resort the matrix according to the same evidence. Only the best dimensions can change during the sorting procedure, but their number can never increase.

The sorting schema requires the active user and item ID, so we can establish a point of reference for the similarity computation. Note that both active vectors must be in the best dimensions before the sorting and they must be the first in their respective lists after the sorting. This requirement ensures that we do not accidentally remove the active vector IDs:

*sort_matrix*

$matrix$

$active_user_id? : ID$

$active_item_id? : ID$

$matrix_size? : \mathbb{N}$

---

$active_user_id? \in best_users$

$active_item_id? \in best_items$

$matrix_size? > 0$

$$uv\,[u : ID] == \{A : ratings \mid A.user_id = u \land A.item_id \in best_items \bullet A.item_id \rightarrow A.rating\}$$

$$iv\,[i : ID] == \{A : ratings \mid A.item_id = i \land A.user_id \in best_users \bullet A.user_id \rightarrow A.rating\}$$

$$best_users' = \{u : best_users, j = 1..\#best_users - 1 \mid SIM(uv[active_user_id?], uv[u(j)]) \geq SIM(uv[active_user_id?], uv[u(j+1)])\}$$

$$best_items' = \{i : best_items, j = 1..\#best_items - 1 \mid SIM(iv[active_item_id?], iv[i(j)]) \geq SIM(iv[active_item_id?], iv[i(j+1)])\}$$

$\#best_users' \leq matrix_size?$

$\#best_items' \leq matrix_size?$

$active_user_id? = best_users'(1)$

$active_item_id? = best_items'(1)$

$users' = users$

$items' = items$

$ratings' = ratings$

$\#best_users' \leq \#best_users$

$\#best_items' \leq \#best_items$

All user and item vectors in this matrix should be organized in order of decreas-

ing similarity to the active vector, according to the best dimensions. The new best users and items should correctly reflect that ordering. To simplify the specification, we define two parameterized shortcuts that look up user and item vectors by their respective IDs. These shortcuts return a portion of the vector, specified by the best dimensions, which guarantees that we sort the matrix according to the specified dimensions. We also reduce the size of the best dimensions according to the *matrix_size* parameter. Note that matrix size must be any positive number with no upper bound. If the matrix is already small enough, the best dimensions will not change. This way, accidentally sorting or resorting a small matrix does not affect it.

Once the matrix has been sorted, we can remove irrelevant data. The purpose of matrix truncation is to reduce the matrix to its final dimensions, specified by best users and items. Note that truncation removes irrelevant users, items, and ratings, but does not introduce any new ones. Also, note that the best dimensions must stay the same during the truncation process. This behavior is defined as follows:

*truncate_matrix*

*matrix*

$users' = best_users$

$items' = best_items$

$ratings' = \{A : ratings \mid A.user_id \in best_users \wedge A.item_id \in best_items\}$

$users' \subseteq users$

$items' \subseteq items$

$ratings' \subseteq ratings$

$best_users' = best_users$

$best_items' = best_items$

Since we keep track of users/items and best users/items separately, the best dimensions may change as we sort and resort the matrix. However, we do not discard any data until we are certain about the final shape of the matrix. Therefore, the truncation procedure should be the last state of the matrix.

To guarantee that the input is generated properly, we establish the correct order of matrix states. The state transition diagram in Figure 6.3 relates the three main states of a matrix as we refine the ratings within it. The initial matrix contains all relevant ratings, which are only sorted if the matrix is large enough. If the matrix is small, sorting accomplishes nothing because no data is removed. If the matrix is large, sorting establishes vector relevance, so the least valuable data may be discarded. If one of the matrix dimensions is small enough, resorting the matrix does not affect the result because rearranging vector dimensions does not change vector similarities. In either case, the matrix is truncated to a uniform size as the last step of the input generation process.

Note that we may want to sort the matrix multiple times. This model of the input generation component supports such behavior. The constraint on the size of the best dimensions will cause the algorithm to resort the vectors according to local similarity. In fact, each subsequent sorting iteration depends on the best dimensions established during the previous iteration, which is the essence of the recursive input generation algorithm.

This model can perform standard and recursive sorting, which is a major requirement for our case study. Table 6.4 outlines the proper course of action for the initial matrix of every shape. In case there are not enough ratings, the algorithm does not

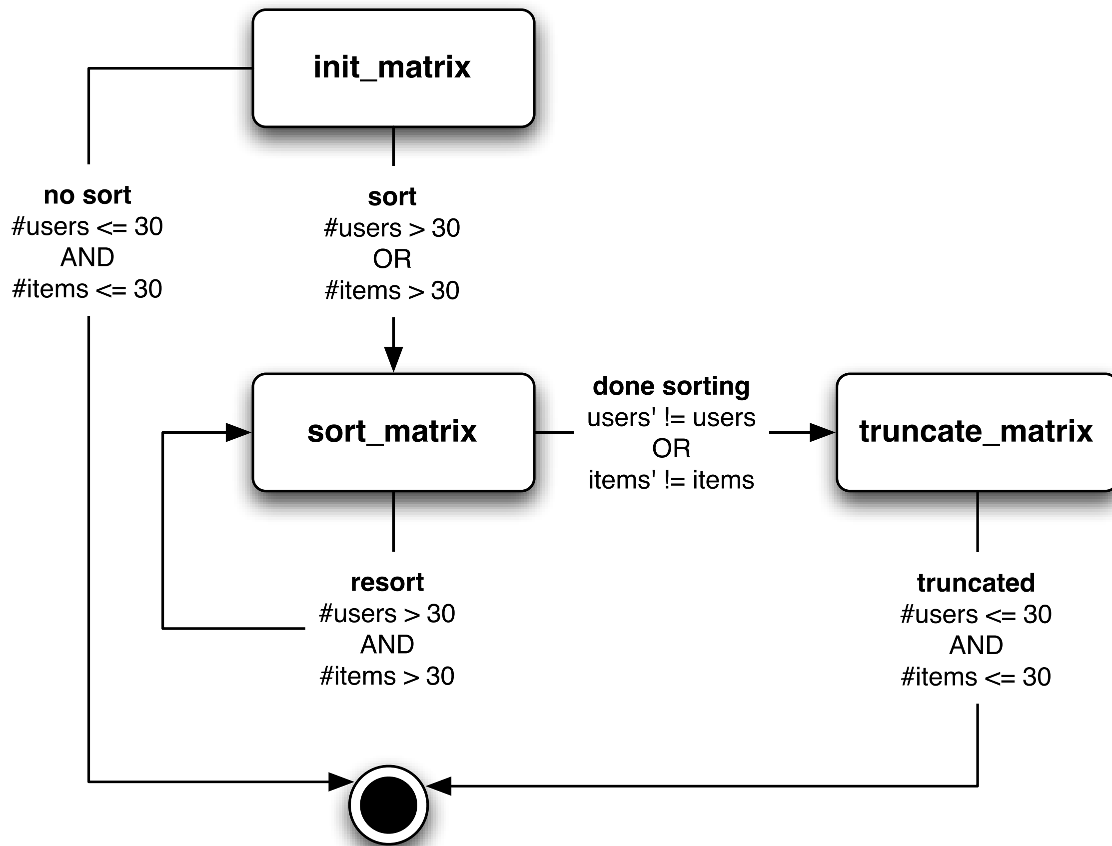


Figure 6.3: Matrix State Changes During the Input Generation Process

apply. Fortunately, there are no such matrices in our dataset. If there are less than 30 users *and* 30 items, we do not sort the matrix. If there are less than 30 users *or* items, we sort the matrix only once. Every large matrix should be sorted twice. This is the only case when our recursive algorithm affects the quality of recommender input, but it applies to 83% of all recommendations.

A sorted and truncated matrix is ready for combination algorithm consumption. The matrix may be serialized or formatted in a way that facilitates matrix transportation to the combination algorithms. However, this responsibility is outside of the scope of the input generation component, which keeps coupling down and en-

| | | #users | | |
|---|---|---|---|---|
| | | <2 | 2..30 | >30 |
| | <2 | N/A | N/A | N/A |
| **#items** | 2..30 | N/A | Done | Standard |
| | >30 | N/A | Standard | Recursive |

Figure 6.4: Matrix Shape Effects on Algorithm Selection

courages reuse. The primary concern of this component is the content of the input matrix. This way, the matrix transportation method can change according to the overall architecture of a recommender system, without affecting the component that produced it.

## 6.2 Static Model Implementation and Analysis

This section presents the implementation details of the data within our input generation component. We use a relational database to store the dataset and produce input matrices. These related tasks require two kinds of database schemas. Permanent relations always exist in the database and their only purpose is to store data. Figure 6.5 shows our permanent database schema. Temporary relations are created on-demand by individual input generation processes. These relations exist only within the context of the executing process and therefore disappear as soon as it stops. Figure 6.6 shows our temporary database schema.

We store the original dataset in the *tbl_ratings* table of three columns: *user_id*, *item_id*, and *rating*. This is the largest table in the entire system, because it takes up over 1.5GB for data and 3GB for index. To speed up vector normalization, we precompute user and item averages in two dedicated tables, *tbl_users* and *tbl_items*.

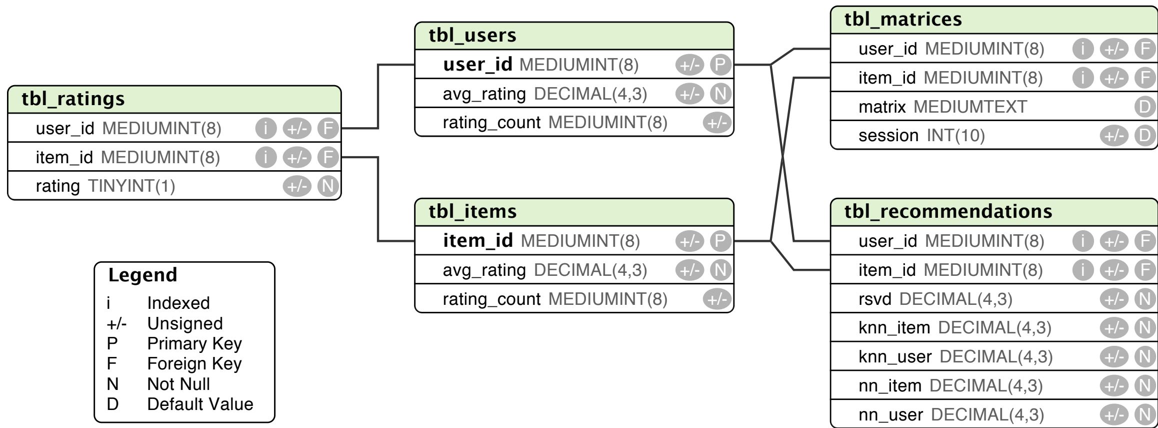


Figure 6.5: Permanent Database Schema

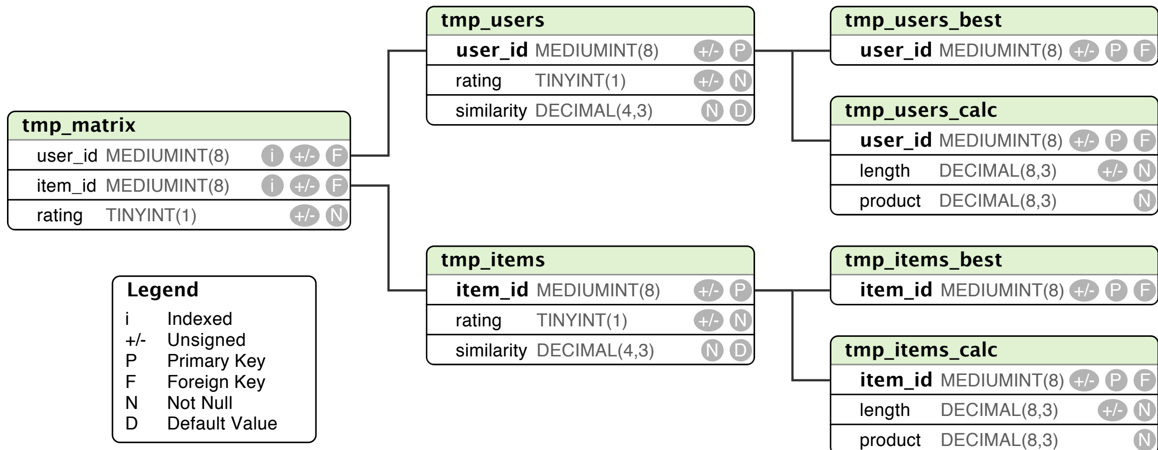


Figure 6.6: Temporary Database Schema

When making recommendations, combination algorithms request a precomputed matrix from the *tbl_matrices* table and mark it as processed in the *session* field. The resulting recommendation is added to the *tbl_recommendations* table under the appropriate field (*rsvd*, *knn_item*, *knn_user*, *nn_item*, or *nn_user*). Once all recommendations are recorded, we can compute the RMSE of individual combination algorithms by comparing their guess in *tbl_recommendations* to an actual rating in *tbl_ratings*.

To generate matrices, we populate the *user_id* and *item_id* fields of the *tbl_matrices* table and launch an instance of the input generation procedure (See Appendix: Listing A). Multiple parallel processes can execute this code on the same machine, which improves scalability. Each simultaneous process uses the *session* field to reserve a matrix it is currently making, thus forcing others to avoid it (See Appendix: Listing B). A finished matrix is cached in the *matrix* field for future consumption.

Each instance works with both database schemas because some of the dataset must be copied to the temporary schema and the finished matrix is stored in the permanent schema. Each instance creates its own copy of the temporary schema (See Appendix: Listing C). As a result, all instances share the permanent schema, while maintaining a duplicate of the temporary one. Because the temporary schema only exists within the session that created it, multiple copies of it could exist independently.

Once the temporary schema is created, an input generation process populates *tmp_user*s with all *user_id*s that rated the active item as well as the actual *rating* that was given. A similar process populates *tmp_items*. The *similarity* field is initialized to zero because no vectors have been loaded yet. To compute vector similarities, we first construct a matrix of ratings by populating *tmp_matrix* with all records from

*tbl_ratings* that contain *user_id* from *tmp_users* and *item_id* from *tmp_items*.

Once we have some ratings in *tmp_matrix*, we can compute the *similarity* of every vector inside it. Due to a MySQL limitation, this process has to be done in two steps, with intermediate results stored in *tmp_users_calc* and *tmp_items_calc*. Each instance uses both tables to determine the similarity between the active vector and its neighbors. Initial sorting is the second slowest step of the process because *tmp_matrix* could get very large for popular items and involved users. However, subsequent sorting should be faster because it is based on a subset of all dimensions captured in *tmp_users_best* and *tmp_items_best* tables.

We employ a consistent naming convention to identify the shared data. Table 6.1 relates the structure of our formal model and its implementation. Every property is implemented by a database relation with an appropriate name. We use prefixes to distinguish between the permanent dataset and temporary matrix implementations. All permanent relations have a *tbl_* prefix and all temporary relations have a *tmp_* prefix. This practice ensures that we store all of the necessary data and that we can easily determine its purpose.

Since our system works with large amounts of data, it is also important to ensure that our data is properly modeled, implemented, and transmitted. Fortunately, the structure of our data is simple, so it is easy to verify that we have modeled it correctly and consistently. The IDs of both users and items are modeled and implemented as positive integers. All ratings are on a discrete scale from one to five and they are modeled and implemented as such.

We also formalized a record data type and we implement it the same way in

| Model | Implementation |
| --- | --- |
| dataset.users | tbl_users |
| dataset.items | tbl_items |
| dataset.ratings | tbl_ratings |
| matrix.users | tmp_users |
| matrix.items | tmp_items |
| matrix.ratings | tmp_matrix |
| matrix.best_users | tmp_users_best |
| matrix.best_items | tmp_items_best |

Table 6.1: Static Model Structure Consistency

| Model | Implementation |
| --- | --- |
| user_id: ℕ, ID>0 | @user_id MEDIUMINT(8) unsigned NOT NULL |
| item_id: ℕ, ID>0 | @item_id MEDIUMINT(8) unsigned NOT NULL |
| rating: ℕ, rating=0..5 | @rating TINYINT(1) unsigned NOT NULL |

Table 6.2: Record Data Type Consistency

both permanent and temporary schemas. Table 6.2 shows more details on our data modeling consistency. The medium integer data type provides a sufficient amount of space to record all IDs. The unsigned property means that only positive numbers may be recorded. The not null property means that the field is required. In other words, there may not be any partial records. This implementation mirrors the formal specification data types.

We also consistently implement our input parameters. Table 6.3 shows the inputs used in the formal specification and their implementation with the same types. The active user/item IDs are modeled and implemented with the same type and range as the data they represent. Therefore, we can address any user/item tuple in the dataset and produce an input matrix for it.

| Model | Implementation |
|---|---|
| active_user_id?: ℕ, ID>0 | @active_user_id unsigned MEDIUMINT(8) |
| active_item_id?: ℕ, ID>0 | @active_item_id unsigned MEDIUMINT(8) |
| matrix_size?: ℕ, matrix_size?>0 | @matrix_size unsigned TINYINT(3) |

Table 6.3: Input Parameter Consistency

## 6.3 Behavioral Model Implementation and Analysis

Our input generation algorithm is the essence of our research, so its formal specification is definitely relevant, especially as a design specification. Even though there may be other designs and various ways to implement them, we present one possible solution that matches our model. To verify its correctness, we compare our model to the final implementation and explain any inconsistencies. We use our formal model to confirm that the implementation corresponds to earlier plans and that all functions are carried out as planned. In this section, we verify our implementation's compatibility, correctness, and completeness.

### 6.3.1 Compatibility with the Rest of the Architecture

The previous chapters presented the input and output interface requirements for the input generation component. The input interface requires each recommendation request to identify a user-item tuple. The *get_matrix* schema models this interface and a procedure implementing it (See Appendix: Listing D). The output interface requires the set of ratings corresponding to at most 30 user and 30 items vectors.

The *truncate_matrix* schema and its implementation satisfy this requirement (See Appendix: Listing F). In other words, we can be sure that the input generation component receives and produces data in the correct format and satisfies the interface requirements imposed by other layers.

There has to be a way to deliver the matrix to the combination algorithms. We implement a matrix serialization procedure that produces a newline-separated string of single digit ratings, where zeros represent missing ratings (See Appendix: Listing G). However, other implementations do not have to rely on this particular format. For example, one may combine our input generation component with a single combination algorithm that works on the matrix directly. As long as the surrounding code agrees on the input generation component interfaces, a recommender system may be built around it. In fact, our formal specification provides all of the necessary information for such integration.

### 6.3.2 Correct Implementation of the Existing Specification

Sometimes the functions implement services with names and interfaces that match, but behaviors that do not. To make sure each function works as required, our formal model includes behavioral specification for the input generation component. The *get_matrix* and *init_matrix* interfaces represent the first two steps of the matrix generation process. This functionality is implemented with a single procedure called *load_matrix* (See Appendix: Listing D). It implements the data access task.

The data processing task is modeled by the *matrix* schema and its two behaviors. We implement the matrix sorting behavior in the *sort_matrix* procedure (See

Appendix: Listing E). It computes vector similarities and keeps track of the best dimensions, thus implementing step three of the input generation process. The *truncate_matrix* procedure reduces the matrix to its final size and implements the last step of the input generation algorithm (See Appendix: Listing F).

Our input generation algorithm can adapt to a number of input selection strategies. For example, it may return a matrix with just the relevant ratings, a matrix that has been sorted only once, i.e., the standard approach, or a matrix that has been sorted multiple times, i.e., the recursive approach. The way we accomplish such configuration flexibility is through separation of concerns that distinguish these approaches. In fact, all three algorithms represent incremental additions of functionality. To ensure efficiency, we implement each behavior separately. This section covers the individual procedure implementations and explains how they satisfy the formal specification.

### Load Matrix Implementation

The *get_matrix* and *init_matrix* schemas identify the read and write operations that locate the relevant ratings. We implement the write and read functionality with a single query using the INSERT ... SELECT construct (See Appendix: Listing D). However, we first truncate the existing tables to make sure that the only data they contain is the data we explicitly add. As a result, we can guarantee that all data selected from the dataset will appear in the matrix. Furthermore, this procedure reads from the dataset and never writes to it, thus satisfying the $\Xi$ *dataset* schema requirements.

The Z and SQL conditions on relevant vectors match. For example, the *init_matrix* schema specifies a restriction of only those user records that contain the active item ID:

$$A : ratings \mid A.item_id = active_item_id?$$

We implement it with an identical restriction in the SQL implementation:

```
1 WHERE item_id=@active_item_id
```

This query also guarantees that each user has at least one rating associated with them.

However, the *get_matrix* schema suggest that there may be one exception to this rule. For example, the active user may not have a rating on the active item, yet the active user should be in the users list. We use the REPLACE query to accomplish two things at once, make sure that the active user ID is in the *tmp_users* table and overwrite the known rating with a zero. Since we cannot remove the user's record, we neutralize its rating by giving it a value that will not participate in the similarity computation:

```
1 REPLACE INTO tmp_users(user_id, rating) VALUES(@active_user_id, 0);
```

Note that the matrix is initialized with the best dimensions as all available dimensions. We copy all relevant user IDs into *tmp_users_best* with the INSERT ... SELECT statement. This procedure satisfies the *matrix* schema requirement because initially the two sets are identical. Furthermore, because we initialize the best dimensions after adding the active vectors to the matrix dimensions, we can be certain that the active vectors are part of the best dimensions. In other words, our code satisfies

the following *sort_matrix* schema requirement:

$$active_user_id? \in best_users$$

Finally, we populate the matrix with relevant ratings. The *get_matrix* schema specifies them as any rating that belongs to a relevant user and a relevant item:

$$A : ratings \mid A.user_id \in users! \land A.item_id \in items!$$

We accomplish this as a dual join of the relevant users and relevant items on the dataset:

```
1  FROM    tbl_ratings ratings
2  JOIN    tmp_items items ON ratings.item_id=items.item_id
3  JOIN    tmp_users users ON ratings.user_id=users.user_id
```

Note that if the active user has rated the active item, their IDs will be in the *tmp_users* and *tmp_items* respectively, so the matrix will contain the active rating. However, active vectors record the active rating as missing, so it will not participate in the similarity computation. Therefore, we can be certain that the integrity of our recommendations has not been compromised.

The way we load the matrix satisfies all conditions of the formal model. The purpose of the load matrix procedure is to make a smaller version of the original dataset, so if the dataset satisfies its requirements, a properly selected subset will too. For example, assuming that the ratings are unique in the original dataset, selecting a subset of them is going to be unique as well. Also, because we load users and items first, each selected rating is guaranteed to belong to a known user and item. As a result, *load_matrix* procedure correctly implements the *get_matrix* and *init_matrix* schemas.

**Sort Matrix Implementation**

The sort matrix procedure contains the logic for deciding whether to sort the matrix (See Appendix: Listing E). Note that the matrix dimensions are equal to the number of ratings for the respective active user and active item. For example, if Alice has a total of 10 ratings and Titanic has a total of 15 ratings, the relevant matrix about her existing opinion about this movie will be exactly 10x15. The rating count information is static for this particular dataset, so there is no need to inspect the matrix. In fact, we can determine whether sorting is necessary just by comparing the precomputed vector rating counts:

```
1  IF (relevant_item_count > @matrix_size OR relevant_user_count > @matrix_size) THEN
2      ...
3      IF (relevant_item_count > @matrix_size AND relevant_user_count > @matrix_size)
           THEN
4          ...
5      END IF
6  END IF
```

Note that the outer condition is only true if one or both of the matrix dimensions are larger than the desired matrix size, which is exactly when the standard approach applies. If neither dimension is large enough, no sorting occurs and the entire relevant dataset is returned. The inner condition is only true when both of the dimensions are greater than the desired matrix size. In that case, the matrix will be sorted one more time, which corresponds to the recursive approach. As a result, the *sort_matrix* procedure correctly implements the state diagram.

This procedure relies on the correct implementation of the two similarity measures, cosine similarity and Pearson's correlation. The formulas for computing the two types

of similarity are as follows:

$$Cosine(a,b) = \frac{\sum_{i \in a \cap b} a_i b_i}{\sqrt{\sum_{i \in a \cap b} a_i^2 \sum_{i \in a \cap b} b_i^2}} = \frac{a \cdot b}{\mid a \mid\mid b \mid}$$

$$Pearson(a,b) = \frac{\sum_{i \in a \cap b} (a_i - \bar{a})(b_i - \bar{b})}{\sqrt{\sum_{i \in a \cap b} (a_i - \bar{a})^2 \sum_{i \in a \cap b} (b_i - \bar{b})^2}} = \frac{a' \cdot b'}{\mid a' \mid\mid b' \mid}$$

Note that both similarity measures share the same form, with the only exception being that Pearson's correlation formula uses relative ratings instead of the actual ones.

To eliminate redundant code, we separate the common parts of the two similarity measures. The *compute_similarities* procedure computes the top and bottom parts of the fraction, i.e., the dot product and magnitudes of the two vectors. These values are different for the two similarity measures (See Appendix: Listings H and I). The *save_similarities* procedure computes the decimal equivalent of the fraction for each vector, saves their similarities, and updates the best dimensions tables (See Appendix: Listing J). These operations are the same for both similarity measures.

The cosine implementation uses the following code to figure out the length and dot product of each neighbor, $b$, with the active vector, $a$:

```
1 SQRT(SUM(POW(b.rating, 2))),
2 SUM(a.rating*b.rating)
```

Pearson's correlation uses the same formula, except the ratings are normalized by that vector's average rating:

```
1 SQRT(SUM(POW(b.rating-c.avg_rating, 2))),
```

```
2 SUM((a.rating-active_user_avg_rating)*(b.rating-c.avg_rating))
```

Both implementations correctly mirror the formula for the magnitude of a neighbor's vector as well as its dot product with the active vector.

Note that these computations depend on the amount of shared ratings and are restricted by the best dimensions. For example, *sort_matrix* schema specifies how the user vectors should be compared:

$$uv\,[u : ID] == \{A : ratings \mid A.user_id = u \wedge A.item_id \in best_items\}$$

Our implementation matches the formal specification:

```
1 FROM    tmp_items as a
2 JOIN    tmp_items_best as best ON a.item_id=best.item_id
3 JOIN    tmp_matrix as b ON a.item_id=b.item_id
4 JOIN    tbl_users as c ON b.user_id=c.user_id
5 WHERE   a.rating > 0
```

Here, *tmp_items* represents the active user vector and *tmp_items_best* table restricts which items participate in the comparison. This list is very small, so it is faster to join it first. However, it is not smaller then *tmp_items* because otherwise there would be no need for sorting. Then we join the *tmp_matrix* table on the same item IDs to produce a set of user vectors across their common dimensions.

We also join *tbl_users* so we can use their rating averages. Finally, we restrict the computation to only positive ratings in this active vector. This restriction explicitly prevents a known active rating from affecting a neighbor's similarity. The last two lines are not necessary for the cosine similarity measure because it does not use vector averages and a zero rating does not affect the dot product nor vector magnitude.

Once all components of the similarity fraction have been computed we store and sort them with the *save_similarities* procedure. It computes the similarity of each user neighbor according to the following formula, in line with the cosine similarity and Pearson's correlation specifications:

```
1 SET a.similarity=b.product/(active_user_length*b.length)
```

The purpose of this procedure is to compute the similarity of every neighbor to the active vector and then record the top vectors as the new best dimensions:

```
1 INSERT INTO tmp_users_best(user_id)
2 SELECT      user_id
3 FROM        tmp_users
4 WHERE       similarity>0
5 ORDER BY    similarity DESC
6 LIMIT       ?
```

Here, the question mark is a placeholder for the *@matrix_size* value. Because we select the best users from the relevant users the resulting list is guaranteed to be a subset of the original:

$$best_users \subseteq users$$

The *sort_matrix* schema also specifies that a limited number of best users must be chosen according to their decreasing similarity:

$$\{SIM(uv[active_user_id?], uv[u(j)]) \geq SIM(uv[active_user_id?], uv[u(j+1)])\}$$

$$\#best_users' \leq matrix_size?$$

We establish the ordering with the ORDER BY clause and specify the size of the list with the LIMIT clause. Therefore, this procedure correctly implements the sorting and dimension reduction properties of our algorithm.

According to the *sort_matrix* schema, the active user and item IDs must be part of the matrix before and after the sorting:

$$active_user_id? \in best_users$$

$$active_user_id? = best_users'(1)$$

The precondition is satisfied by *load_matrix* procedure, which explicitly includes them. The postcondition is satisfied by the *save_similarities* procedure. By definition, the cosine similarity and Pearson's correlation of a vector compared to itself must be 1, which is the maximum similarity:

$$w(a,a) = \frac{\sum_{i \in a} a_i a_i}{\sqrt{\sum_{i \in a} a_i^2}\sqrt{\sum_{i \in a} a_i^2}} = \frac{\sum_{i \in a} a_i^2}{\sum_{i \in a} a_i^2} = 1$$

Note that because the active vector has the maximum possible similarity, it is guaranteed to be the first of the best dimensions, since they are chosen in order of decreasing similarity. Therefore, we can guarantee that the active vectors will always be part of the matrix.

### Truncate Matrix Implementation

The truncate matrix implementation is a little different than its specification. Instead of truncating and populating the users and items of the matrix like we have done before, we delete those IDs that are not in the best dimensions (See Appendix: Listing F). We keep vector similarities of the final matrix because such information may be useful for serializing it (See Appendix: Listing G). Finally, we repopulate an empty matrix with only the best dimensions to satisfy the final matrix requirement.

Doing so is a faster and more reliable way of loading ratings. The constraints on the *truncate_matrix* schema and its implementation are identical:

$$ratings' = \{A : ratings \mid A.user_id \in best_users \wedge A.item_id \in best_items\}$$

```
1 INSERT INTO tmp_matrix(user_id, item_id, rating)
2 SELECT      users.user_id, items.item_id, ratings.rating
3 FROM        tbl_ratings ratings
4 JOIN        tmp_items_best items ON ratings.item_id=items.item_id
5 JOIN        tmp_users_best users ON ratings.user_id=users.user_id;
```

### 6.3.3 The Rao Blackwell Theorem Implementation

We also correctly implement the Rao Blackwell theorem, i.e., two passes of the sorting algorithm that establish the sufficient statistic and improved local similarity weights. Note that *load_matrix* procedure initializes the best dimensions to the initial dimensions of the matrix. In other words, the first iteration of the sort procedure compares vectors according to all dimensions of the matrix, i.e., global similarity. Also note that the best dimensions are not modified until after the similarity has been computed, which means that the best dimensions will be restricted on the second iteration, i.e., local similarity.

The same local similarity is used in the combination algorithm, except the combination algorithm computes it from the serialized matrix instead of the database. However, this local similarity does not depend on just any set of dimensions, but rather the best dimensions that were chosen according to the their global similarity ranking. Therefore, we can say that the local similarities of the final matrix have been conditioned by the global similarity. According to the Rao Blackwell theorem,

the expected value of the neighbor's opinions weighted by such local similarities will be better than the opinions weighted by the local similarities computed across any other set of common dimensions.

## 6.4 Conclusion

This chapter presents a more explicit model of the input generation component. We use the Z notation to describe the model and identify the restrictions that must be met at each stage of the input generation process. We establish the procedure interfaces, their internal behavior, and legal interactions. We also address the input, intermediate results, and output data for our input generation component. The formal model of the system behavior provides the essential details, enhances our understanding, and serves as a blueprint for implementation. We also evaluate the quality of our input generation model and its implementation. We validate the correctness, completeness, and compatibility of our model to justify the implementation that meets all of our system requirements.

# Chapter 7

# Evaluation of Recommendation Accuracy

In order to research ways to improve recommendation accuracy, we develop a distributed architecture that allows quick accuracy evaluations with no rework. Our architecture consists of a single input generation component and multiple combination algorithms. Previous chapters outlined our input generation component, the combination algorithms, and dataset properties. This chapter evaluates the benefits of combination algorithm tuning, data normalization, and input selection. Our goal is to find out whether the latter practice improves recommendation accuracy more than the rest.

Our research method is fairly standard across the available literature. First, we build the framework to import and manage the dataset. Details of our data management component are described in the previous chapter. We also create routines to calculate RMSE values for our predictions. Our reasons for choosing this particu-

lar metric are available in the first chapter. Our experiments evaluate recommender accuracy with this measure and investigate the techniques that minimize it.

The experiments are organized in a particular order. First, we implement different combination algorithms and generate standard input recommendations for each one. We then try different variations of the algorithms and compare the resulting RMSE values. We tune combination algorithms to maximize their accuracy. Once tuned algorithms are in place, we evaluate the benefits of data normalization and recommendation combination/aggregation. Finally, we combine the most accurate recommender configurations with a set of input generation component implementations. Each stage of the evaluation procedure introduces only one new method of improving accuracy so we can easily identify its benefit.

## 7.1 Evaluation Assumptions

Rating an item on a numerical scale is an enormous simplification of a complex human phenomenon. However, for the purposes of this research, we assume that each rating is an accurate indication of the user's true opinion about an item. In other words, if a person rated one item as five stars and another as four stars, we assume he/she liked the first item more. Even though each rating is highly personal and could be determined by a myriad of related stimuli, we assume that the ratings may be accurately determined by a community of similar users who have rated similar items.

Our analysis of the dataset shows one user who submitted 17,653 ratings, which is 47 votes short of rating every single movie in the dataset. These votes are unlikely

to be the actual opinion of a single person. It is probably a base approximation of the Netflix recommender. However, for the purposes of this research we assume that this user profile is completely legitimate. We also assume that all ratings are real, i.e., submitted by actual people. Likewise, we assume that rating patterns and biases are the result of user preferences and are not influenced by other aspects, e.g., user interface. Therefore, we assume that the dataset is an accurate representation of the actual user opinions.

## 7.2 Accuracy Goals

To establish a point of reference for our experiments, we examine some of the well-known results from the Netflix website, www.netflixprize.com. It lists the typical prediction errors of many trivial recommendation approaches that suggest the same rating for every item. For instance, recommending a four star rating for each movie is the most accurate (RMSE = 1.1748), because each recommendation is close to the overall average rating of 3.6 stars. Likewise, recommending 3.6 stars for everyone gives an even smaller error of 1.1287. This value may be reduced further by recommending the movie or user average for each movie and user request. This results in a typical error of 1.0533 for an average movie and 1.0651 for an average user approach. In general, any recommender that is consistently off by one or more units is considered inferior. These figures establish the lowest accuracy threshold below which the recommendations are no longer useful.

However, the highest accuracy threshold is still largely unknown. Thousands of

contestants spent years trying to reduce a typical error of their recommender on the Netflix dataset. On July 26th, 2009 a team named “BellKor’s Pragmatic Chaos” reached a previously impossible RMSE value of 0.8567. The authors of the winning algorithm published three papers detailing their approach [81,127,160]. Their work has motivated our research by showing that a significant improvement in recommendation accuracy is possible on large and sparse datasets. We believe that recommendation accuracy can be improved even further. We try to get closer to the lowest possible error, dictated by the unpredictable human nature.

## 7.3 Combination Algorithm Tuning Effects on Recommendation Accuracy

Studies of Web usability show that user satisfaction depends on the service latency. Ideally the recommendation is generated in under one second, which would result in a truly interactive user experience [137]. However, this kind of performance is difficult to achieve. Therefore, we do not consider configurations that take over five seconds to produce a single recommendation, because they are too slow for practical purposes [105]. In fact, we precompute input matrices to produce quick recommendations with consistent latency, because each algorithm consumes small matrices of fixed size.

To tune combination algorithms, we repeatedly produced 1,000 random recommendations with different parameters. For this experiment we used the cosine standard input generation approach with a $30 \times 30$ truncation. In other words, each

matrix contains at most 30 users and items as determined by the standard cosine similarity. Previous work suggests using no more than 30 neighbors when making a recommendation. Our experiments confirm that a neighborhood of this size produces the most accurate suggestions.

We use three different combination algorithms in our recommender system prototype: K Nearest Neighbors (KNN), Robust Singular Value Decomposition (RSVD), and Neural Network (NN). The KNN and NN methods have two subtypes, user-based and item-based [95, 164]. The subtype of a method depends on how the input matrix is interpreted. By default, rows in a matrix represent item vectors. However, rows represent user vectors in a transposed matrix. Therefore, if the recommendation was done on the original matrix, we refer to an algorithm as item-based and if the matrix was rotated, we call it user-based. The RSVD algorithm does not have any subtypes because its purpose is to estimate the entire matrix, with no distinction between rows and columns. Therefore, we compare the root mean squared errors of five algorithms: RSVD, KNN-Item, KNN-User, NN-Item, and NN-User. The results help us formulate and justify final algorithm configurations.

### 7.3.1 Robust Singular Value Decomposition Recommender Model

The RSVD algorithm approximates an incomplete matrix of ratings from data within it. It locates a set of numbers within two composite matrices, the product of which produces an estimate that most closely resembles the set of known ratings. Unknown ratings are estimated within the same matrix. We tuned three parameters within this

approach: the learning rate, the number of features, and the number of iterations.

The number of iterations represents how many times the approximation will be recalculated. Since cells of a matrix are adjusted to reduce the approximation error with each iteration, a large number of iterations is not efficient, unless the learning rate is sufficiently small. The learning rate controls how fast the cells of the approximated matrix change. It also affects how many iterations are necessary to complete a satisfactory estimate. If the learning rate is large, the estimation is fast, but it is more likely to get out of local minima on the approximation error surface. A large learning rate will dramatically change the estimates, thus jumping around the estimation error surface. If the learning rate is small, the estimation is slow, but it is more likely to converge. Therefore, gradually decreasing the learning rate is often required to achieve continuously improving estimates. We consider this adjustment in one of the following experiments.

The number of features affects the dimensions of two composite matrices. More features improve accuracy because they provide more space for estimating rating patterns. In fact, some authors have successfully predicted the entire Netflix dataset in a single RSVD estimate with over 96 features [121]. However, we cannot set the number of features too high because such a model would take too long to train. Since our combination algorithms reside in a browser, we do not have the computational resources to train that many features. Therefore, we have to reduce the number of features to achieve appropriate performance.

We start with a static learning rate to find out the most optimal number of features and cycles. In general, a small learning rate and a large number of iterations

is necessary to achieve the best estimate. We test this claim with a 0.02 learning rate. Figure 7.1 shows that the best estimate contained only two features over 10 cycles (RMSE = 0.9785). Our RSVD algorithm estimates small matrices with at most 900 cells, so it does not require a large number of features or learning cycles. Considering a higher number of features and cycles did not produce interactive results. This configuration approximates the rating matrix quickly, and the small learning rate helps the algorithm converge to an accurate estimate.

| | | Cycles | | | |
|---|---|---|---|---|---|
| | | **10** | **50** | **100** | **200** |
| **Features** | **1** | 0.9837 | 0.9845 | 0.9845 | 0.9845 |
| | **2** | 0.9785 | 1.0212 | 1.0265 | 1.0235 |
| | **3** | 0.9842 | 1.0861 | 1.1389 | 1.1970 |
| | **4** | 0.9907 | 1.1518 | 1.3335 | 1.4015 |
| | **5** | 1.0062 | 1.1896 | 1.3337 | 1.5773 |

Figure 7.1: RSVD Features and Cycles Tuning

Our next experiment identified the most optimal learning rate and confirmed that only 10 iterations are sufficient to approximate a matrix. We considered six different configurations of the RSVD algorithm with one and two features over 10, 25, and 50 learning cycles, because these configurations showed the most promising results in the previous experiment. Figure 7.2 shows that an RSVD configuration with two features over 10 cycles and a 0.01 static learning rate produces the lowest error (RMSE = 0.9688).

Our previous experiments only used a static learning rate, but reducing the learning rate with each cycle could produce better results. A large learning rate covers a greater number of close estimates and a small learning rate can find the closest of

| | | Features x Cycles | | | | | |
|---|---|---|---|---|---|---|---|
| | | **1x10** | **1x25** | **1x50** | **2x10** | **2x25** | **2x50** |
| **Static Learning Rate** | **0.01** | 0.9726 | 0.9712 | 0.9715 | 0.9688 | 0.9713 | 1.0047 |
| | **0.02** | 0.9842 | 0.9840 | 0.9844 | 0.9801 | 1.0171 | 1.0248 |
| | **0.03** | 0.9997 | 1.0006 | 1.0006 | 1.0536 | 1.0422 | 1.0256 |
| | **0.04** | 1.0185 | 1.0192 | 1.0193 | 1.0483 | 1.0385 | 1.0271 |

Figure 7.2: RSVD Static Learning Rate Tuning

each one. For the final experiment we considered eight different configurations that performed best in the previous experiment. Figure 7.3 illustrates the relative benefit of progressively reducing the learning rate. The results show that the best configuration involves two features over 25 cycles with 0.01 initial learning rate reduced by 15% with each cycle (RMSE = 0.9657). However, our final RSVD configuration includes one feature over 25 cycles with a 0.02 initial learning rate reduced by 25% with each cycle (RMSE = 0.9660). We chose a different set of parameters as our final RSVD combination algorithm because it produces faster results with comparable accuracy.

| | | Features x Cycles x Learning Rate | | | | | |
|---|---|---|---|---|---|---|---|
| | | **1x10x0.02** | **1x25x0.02** | **2x10x0.01** | **2x25x0.01** | **2x10x0.02** | **2x25x0.02** |
| | **0%** | 0.9842 | 0.9840 | 0.9688 | 0.9713 | 0.9801 | 1.0171 |
| **Learning Rate Reduction %** | **5%** | 0.9716 | 0.9713 | 0.9683 | 0.9689 | 0.9694 | 0.9894 |
| | **10%** | 0.9698 | 0.9676 | 0.9713 | 0.9667 | 0.9691 | 0.9688 |
| | **15%** | 0.9681 | 0.9671 | 0.9762 | 0.9657 | 0.9696 | 0.9661 |
| | **20%** | 0.9710 | 0.9677 | 1.0027 | 0.9687 | 0.9703 | 0.9677 |
| | **25%** | 0.9764 | 0.9660 | 1.0059 | 0.9670 | 0.9704 | 0.9658 |

Figure 7.3: RSVD Learning Rate Reduction Tuning

### 7.3.2 K Nearest Neighbors Recommender Method

The KNN algorithm produces recommendations as a weighted average of neighbors' ratings. It has only two parameters, K and N. K adjusts extremely optimistic/pessimistic estimates and N limits the number of neighbors to consider. A greater number

of neighbors does not necessarily improve accuracy. In fact, the recommended number of neighbors is 30 [142]. We tested two subtypes of the KNN algorithm and recorded RMSE scores of each approach in Figures 7.4 and 7.5.

| | | K | | | | | |
|---|---|---|---|---|---|---|---|
| | | **0.5** | **0.625** | **0.75** | **0.875** | **1** | **1.5** |
| **N** | **10** | 0.9719 | 0.9680 | 0.9681 | 0.9725 | 0.9833 | 1.0523 |
| | **20** | 0.9686 | 0.9620 | 0.9591 | 0.9593 | 0.9634 | 1.0126 |
| | **30** | 0.9685 | 0.9617 | 0.9587 | 0.9574 | 0.9603 | 1.0035 |

Figure 7.4: KNN-Item Tuning

| | | K | | | | | |
|---|---|---|---|---|---|---|---|
| | | **0.5** | **0.625** | **0.75** | **0.875** | **1** | **1.5** |
| **N** | **10** | 1.0228 | 1.0069 | 0.9942 | 0.9851 | 0.9795 | 0.9934 |
| | **20** | 1.0184 | 1.0017 | 0.9873 | 0.9762 | 0.9685 | 0.9725 |
| | **30** | 1.0195 | 1.0019 | 0.9874 | 0.9760 | 0.9680 | 0.9696 |

Figure 7.5: KNN-User Tuning

As expected, both subtypes produce their lowest errors at N=30. Therefore, it is not necessary to control N, but rather process the entire matrix. The KNN-User needs no adjustment, with K = 1 (RMSE = 0.9680). However, the KNN-Item approach will naturally produce recommendations that are too optimistic, so we set K = 0.875 (RMSE = 0.9574).

Consistent with previously reported results, the KNN-Item is more accurate than the KNN-User approach because item vectors usually have more ratings than user vectors. Therefore, vector averages, which are used in KNN algorithms, will be more accurate for item-based approaches. Furthermore, the KNN-Item approach is the best of all five algorithms we consider. This proves that a combination algorithm does not need to be complex to be accurate.

### 7.3.3 Neural Network Recommender Model

The NN approach predicts unknown ratings by classifying a vector into one of the previously learned patterns. Both subtypes of this approach have a standard architecture, with an optional hidden layer, five binary output nodes, and a variable number of input nodes. Five binary nodes are required to represent a single rating on a scale from one to five. We examine effects of three tuning parameters: the number of nodes in a hidden layer (0 meaning no hidden layer), the number of training cycles, and the learning rate.

The number of input nodes depends on the amount of input vectors. For instance, when creating a network for a matrix with 10 rows and 20 columns, 95 binary input nodes would be created (five for each of the 19 inputs). Once the network is constructed, it would receive 9 training cases and expected responses. Once the network was trained, it would accept an active vector as input and produce a recommendation by exciting the output node associated with the recommended rating.

Ideally, only one output node would light up in a trained network. However, when the network could not reliably classify the active vector, multiple output nodes lit up, some brighter than others. In such cases, we can reach a decision by choosing the brightest node and ignoring the rest, or by taking a weighted average of all output nodes. Because our experiments show that a weighted average decision produced recommendations that were 20% more accurate, the following experiments use this type of conclusion.

Our first experiment identified the most appropriate learning rate and amount of training cycles. Figure 7.6 shows that the NN-Item works best with 0.1 learning rate

at 50 training cycles (RMSE = 1.024) and Figure 7.7 demonstrates that the NN-User should be used with the same learning rate but 30 training cycles (RMSE = 1.012). Note that the NN-User consistently produces better recommendations than the NN-Item, which is the opposite in the KNN approach. This is because a typical movie has hundreds of viewers, so there are usually many user vectors in a matrix, i.e., more training cases. On the other hand, a typical user has seen fewer movies, so there are not enough item vectors to sufficiently train the network.

Our next experiment isolated the benefit of having a hidden layer in a neural network. Figures 7.8 and 7.9 show the RMSE results of the two types of NN algorithms with 10 nodes in the hidden layer. Adding 15 or more nodes in the hidden layer produced results that were too slow. The NN-Item works best with 0.1 learning rate at 40 training cycles (RMSE = 1.062), which is actually worse than the previous configuration with no hidden nodes. The NN-User should be used with the same learning rate, but 40 training cycles (RMSE = 0.987). In fact, additional hidden nodes in the NN-User increased training time, but caused a significant improvement in the recommendation quality.

Finally, we consider the effects of reducing the learning rate with each training cycle. Figures 7.10 and 7.11 show that there is no benefit to reducing the learning rate for either type of NN. The final configuration for the NN-Item algorithm includes no hidden nodes with a 0.1 static learning rate over 50 iterations. The final configuration for the NN-User algorithm includes 10 hidden nodes with a 0.1 static learning rate over 40 iterations.

| | | Cycles | | | | |
|---|---|---|---|---|---|---|
| | | 20 | 30 | 40 | 50 | 60 |
| Static Learning Rate | 0.05 | 1.049 | 1.035 | 1.036 | 1.030 | 1.033 |
| | 0.10 | 1.042 | 1.042 | 1.027 | 1.024 | 1.035 |
| | 0.20 | 1.043 | 1.031 | 1.044 | 1.029 | 1.032 |
| | 0.30 | 1.035 | 1.048 | 1.029 | 1.034 | 1.033 |
| | 0.40 | 1.040 | 1.046 | 1.036 | 1.042 | 1.036 |

Figure 7.6: NN-Item with No Hidden Nodes Tuning

| | | Cycles | | | | |
|---|---|---|---|---|---|---|
| | | 20 | 30 | 40 | 50 | 60 |
| Static Learning Rate | 0.05 | 1.017 | 1.037 | 1.031 | 1.041 | 1.0339 |
| | 0.10 | 1.024 | 1.012 | 1.030 | 1.030 | 1.0232 |
| | 0.20 | 1.031 | 1.029 | 1.028 | 1.027 | 1.0267 |
| | 0.30 | 1.016 | 1.025 | 1.019 | 1.037 | 1.0275 |
| | 0.40 | 1.023 | 1.041 | 1.045 | 1.031 | 1.0153 |

Figure 7.7: NN-User with No Hidden Nodes Tuning

| | | Cycles | | |
|---|---|---|---|---|
| | | 30 | 40 | 50 |
| Static Learning Rate | 0.10 | 1.078 | 1.062 | 1.068 |
| | 0.20 | 1.071 | 1.074 | 1.092 |
| | 0.30 | 1.078 | 1.094 | 1.086 |

Figure 7.8: NN-Item with 10 Hidden Nodes Tuning

| | | Cycles | | | |
|---|---|---|---|---|---|
| | | 20 | 30 | 40 | 50 |
| Static Learning Rate | 0.05 | 1.010 | 1.011 | 1.001 | 0.999 |
| | 0.10 | 1.001 | 0.994 | 0.987 | 1.001 |
| | 0.20 | 0.993 | 1.014 | 1.037 | 1.028 |
| | 0.30 | 1.016 | 1.049 | 1.065 | 1.048 |

Figure 7.9: NN-User with 10 Hidden Nodes Tuning

## 7.4 Input/Output Tuning Effects on Recommendation Accuracy

Data normalization can make vectors more similar and reduce recommendation errors. In order to compensate for the different ways users rate movies, we normalize ratings in a matrix by vector average. Results in Figure 7.12 show that normalizing the data by item average produces more accurate results because item averages are usually better predictors [13]. However, normalized matrices produce less accurate recommendations than the original ones. This could be because the KNN approach already has data normalization functionality and RSVD fails to accurately estimate a matrix with small values. Figure 7.12 does not contain Neural Network recommenders because they are designed to receive a matrix of discreet values so normalized ratings could not be accepted.

Another way to normalize the matrix is to preprocess it with the RSVD algorithm. Because the RSVD model can estimate multiple missing ratings at the same time, it is often used to normalize input data for other recommenders [23]. Figure 7.13 shows that preprocessing the input matrix makes recommendations worse for every algorithm. This could be because preprocessed matrices no longer represent actual user ratings, but rather an estimate of those ratings. The matrices are less sparse, so each algorithm has more available data. However, the lack of actual ratings makes it difficult for any algorithm to accurately model rating patterns.

To ensure that each algorithm produces a recommendation that falls within the rating scale, we adjust the outputs that are less than one and greater than five on

| | | Hidden Nodes x Cycles x Learning Rate | | | | |
|---|---|---|---|---|---|---|
| | | **0x50x0.1** | **0x50x0.2** | **0x40x0.3** | **0x40x0.1** | **10x40x0.1** |
| | **0%** | 1.024 | 1.029 | 1.029 | 1.027 | 1.062 |
| **Learning Rate** | **5%** | 1.028 | 1.040 | 1.037 | 1.028 | 1.056 |
| **Reduction %** | **10%** | 1.053 | 1.030 | 1.026 | 1.060 | 1.081 |
| | **15%** | 1.069 | 1.045 | 1.034 | 1.058 | 1.084 |
| | **20%** | 1.082 | 1.035 | 1.029 | 1.084 | 1.089 |
| | **25%** | 1.127 | 1.061 | 1.046 | 1.128 | 1.096 |

Figure 7.10: NN-Item Learning Rate Reduction Tuning

| | | Hidden Nodes x Cycles x Learning Rate | | | | |
|---|---|---|---|---|---|---|
| | | **0x30x0.1** | **0x30x0.2** | **10x40x0.1** | **10x40x0.2** | **10x20x0.2** |
| | **0%** | 1.012 | 1.029 | 0.987 | 1.037 | 0.993 |
| **Learning Rate** | **5%** | 1.028 | 1.028 | 1.011 | 0.992 | 0.994 |
| **Reduction %** | **10%** | 1.030 | 1.026 | 1.029 | 1.007 | 0.998 |
| | **15%** | 1.078 | 1.027 | 1.031 | 1.009 | 1.008 |
| | **20%** | 1.092 | 1.046 | 1.042 | 1.013 | 1.016 |
| | **25%** | 1.135 | 1.062 | 1.061 | 1.017 | 1.019 |

Figure 7.11: NN-User Learning Rate Reduction Tuning

| | **RSVD** | **KNN-Item** | **KNN-User** |
|---|---|---|---|
| **No Normalization** | 0.9659 | 0.9575 | 0.9679 |
| **Active User Average** | 1.1941 | 1.2865 | 1.3056 |
| **Active Item Average** | 1.1530 | 1.2527 | 1.3005 |

Figure 7.12: Rating Normalization Benefit

| | **RSVD** | **KNN-Item** | **KNN-User** | **NN-Item** | **NN-User** |
|---|---|---|---|---|---|
| **No RSVD** | 0.9685 | 0.9575 | 0.9679 | 1.0405 | 1.0010 |
| **RSVD** | 0.9878 | 0.9620 | 0.9651 | 1.2996 | 1.2996 |

Figure 7.13: RSVD Preprocessing Benefit

a one to five scale. Figure 7.14 shows that the KNN algorithm is the only one that does not benefit from this, because its recommendations fall within the rating scale by design. For other algorithms, 80% of the time a recommendation that is outside the rating scale is accurate when capped. In the remaining 20%, the over-confident recommendation is wrong, so adjusting it actually reduces the recommendation error.

| | RSVD | KNN-Item | KNN-User | NN-Item | NN-User |
|---|---|---|---|---|---|
| **Uncapped** | 0.968 | 0.958 | 0.968 | 1.049 | 0.995 |
| **Capped [1..5]** | 0.966 | 0.957 | 0.968 | 1.016 | 0.993 |

Figure 7.14: Recommendation Cap Benefit

Recommender systems often provide a discrete recommendation on a discrete rating scale. To accomplish this, we test recommendation rounding benefits. Figure 7.15 shows that, similar to the previously published research [29], rounding off recommendations hurts prediction accuracy. The error is especially high for the NN approach because rounding decreases the benefit of a weighted average conclusion used in this algorithm.

| Rounding Threshold | RSVD | KNN-Item | KNN-User | NN-Item | NN-User |
|---|---|---|---|---|---|
| **None** | 0.9685 | 0.9575 | 0.9679 | 1.0150 | 0.9952 |
| **0.1** | 0.9872 | 0.9763 | 0.9978 | 1.0550 | 0.9997 |
| **0.2** | 0.9906 | 0.9768 | 0.9988 | 1.0587 | 1.0032 |
| **0.3** | 0.9927 | 0.9770 | 1.0004 | 1.0789 | 1.0160 |
| **0.4** | 0.9963 | 0.9859 | 1.0034 | 1.0563 | 1.0311 |
| **0.5** | 1.0095 | 0.9967 | 1.0161 | 1.0770 | 1.0394 |

Figure 7.15: Recommendation Rounding Benefit

Our experiments show that KNN-Item approach is better than KNN-User. This reflects the conclusion by Sarwar et al., who showed that item-oriented collaborative filtering recommendations are more accurate [142]. Bell and Koren also support this

claim with their extensive work on the Netflix dataset [13, 81]. The authors report a RMSE value of 0.9157 for user-oriented and 0.9086 for item-oriented methods. Most importantly, they suggest that the two methods are not mutually exclusive and are expected to produce better results when combined together. In fact, the authors report a slight decrease in RMSE value to 0.9030 by taking a very simple combination of the two suggestions with static weights [13]. Furthermore, Bell and Koren suggest that a more sophisticated combination algorithm could potentially produce even better results.

We test this hypothesis and record the results in Figure 7.16. The upper section contains individual algorithm RMSE scores. The lower section contains the RMSE scores of the minimum, maximum, median, and mean of the five available suggestions. Averaging all recommendations produces the best results, but it is less accurate than the individual KNN-Item predictions.

| | **RMSE** |
|---|---|
| **RSVD** | 0.982 |
| **KNN-Item** | 0.963 |
| **KNN-User** | 0.973 |
| **NN-Item** | 1.058 |
| **NN-User** | 0.998 |
| **MIN** | 1.053 |
| **MAX** | 1.052 |
| **MEDIAN** | 0.989 |
| **MEAN** | 0.976 |

Figure 7.16: Recommendation Aggregation Benefit

We also consider every possible pair of recommender combinations in Figure 7.17. The KNN-Item and KNN-User algorithms make the best pair, but they are less accurate than the individual KNN-Item predictions. However, combining KNN-Item

and NN-User with 4.555 and 0.78 coefficients, produces an RMSE of 0.9614, which is the best score thus far. Similar to previous research [14, 56, 107], combining item and user information has shown better quality recommendations.

| | RMSE |
|---|---|
| KNNI-KNNU | 0.9669 |
| KNNI-NNI | 0.9881 |
| KNNI-NNU | 0.9678 |
| KNNI-RSVD | 0.9692 |
| KNNU-NNI | 0.9928 |
| KNNU-NNU | 0.9686 |
| KNNU-RSVD | 0.9748 |
| NNI-NNU | 0.9986 |
| NNI-RSVD | 0.9984 |
| NNU-RSVD | 0.9737 |

Figure 7.17: Recommendation Combination Benefit

In order to form a baseline of the range of the achievable accuracy, we manually choose the best combination algorithm for 1,000 recommendations. If we somehow know the best algorithm for each recommendation, we can reach an RMSE of 0.761. Choosing among the minimum, average, and maximum of five recommenders produces an RMSE of 0.763. Choosing the best between the minimum and the maximum of the five recommendations produces an RMSE of 0.770. These assumptions show us great potential for recommendation accuracy. However, there is no correlation between the minimum/maximum recommendation and the errors they produce. Furthermore, different algorithms may be considered best under similar circumstances, so there is no reliable way to reach these RMSE scores.

## 7.5 Input Data Selection Effects on Recommendation Accuracy

Previous sections demonstrate that even though some combination algorithms are better than others, tuning them has little effect on recommendation accuracy. We believe that the substance of input data affects recommendation accuracy the most. Therefore, our final solution does not include rating normalization or RSVD preprocessing, but we restrict the recommendation value to the rating scale without rounding it. This section shows how the choice of an input matrix generation algorithm affects recommendation accuracy.

To improve recommendation performance, we consider some faster alternatives to generating the input rating matrix. For example, instead of calculating the similarity of vectors across the entire matrix and then choosing the most similar ones, we pick the first 30 vectors with the most ratings. We can also select 30 vectors with the most similar rating counts and the most similar average rating. These methods are faster than the standard approach because they include less similarity computations, which is the second slowest step of the matrix generation process. Figure 7.18 shows that none of them produce better results.

| | RSVD | KNN-Item | KNN-User | NN-Item | NN-User |
|---|---|---|---|---|---|
| **Standard Approach** | 0.967 | 0.957 | 0.968 | 1.040 | 0.997 |
| **Highest Rating Count** | 1.134 | 1.138 | 1.140 | 1.151 | 1.199 |
| **Similar Rating Count** | 1.690 | 1.018 | 1.074 | 1.237 | 1.085 |
| **Closest Average Rating** | 1.478 | 1.024 | 1.040 | 1.110 | 1.042 |

Figure 7.18: Faster Methods of Selecting Recommender Input

To improve recommendation accuracy, we considered three alternatives to the

standard similarity method. Figure 7.19 shows that using Pearson's correlation is considerably better than using cosine similarity. Also, weighting properties did not improve recommendation accuracy. However, the Pearson's Recursive approach significantly reduced the RMSE score for all recommenders. In fact, reselecting top vectors according to their local similarity was more accurate for both similarity metrics. Finally, anything over two iterations of the Pearson's Recursive algorithm did not improve accuracy.

| Method | RSVD | KNN-Item | KNN-User | NN-Item | NN-User |
|---|---|---|---|---|---|
| Cosine Standard | 1.291 | 1.301 | 1.305 | 1.344 | 1.317 |
| Pearson's Standard | 0.867 | 0.786 | 0.826 | 1.144 | 0.868 |
| | | | | | |
| Pearson's - Pearson's Weighted | 1.805 | 1.318 | 1.319 | 1.384 | 1.372 |
| Pearson's - Cosine Weighted | 1.787 | 1.306 | 1.306 | 1.362 | 1.365 |
| Cosine - Pearson's Weighted | 1.806 | 1.314 | 1.310 | 1.369 | 1.363 |
| Cosine - Cosine Weighted | 1.805 | 1.326 | 1.321 | 1.384 | 1.372 |
| | | | | | |
| Cosine Recursive (2) | 0.923 | 0.980 | 0.945 | 1.164 | 0.998 |
| Cosine Recursive (3) | 0.985 | 0.975 | 0.989 | 1.177 | 1.008 |
| | | | | | |
| Pearson's Recursive (2) | 0.665 | 0.423 | 0.465 | 1.010 | 0.611 |
| Pearson's Recursive (3) | 0.762 | 0.465 | 0.528 | 0.949 | 0.623 |
| Pearson's Recursive (4) | 0.722 | 0.456 | 0.519 | 0.938 | 0.600 |

Figure 7.19: More Accurate Methods of Selecting Recommender Input

## 7.6 Recommendation Accuracy of the Final Prototype Configuration

Our final recommender system chooses its ratings according to two passes of the Pearson's Recursive method. The first pass computes global similarities across user and item vectors. It also records the top 30 most similar users and items. This is

equivalent of establishing a list of expert users and most informative items [37]. The second pass recalculates local user and item similarities, but only according to the top items/users established in the first pass. The final input matrix contains the most relevant ratings provided by the best users on the best items. The list below summarizes the final configuration options of our recommender system prototype.

1. RSVD: [features=1, cycles=25, learning rate=0.02, learning rate reduction=0.25]
2. KNN-Item: [K=0.875]
3. KNN-User: [K=1]
4. NN-Item: [hidden nodes=0, training cycles=50, learning rate=0.1]
5. NN-User: [hidden nodes=10, training cycles=40, learning rate=0.1]

To ensure the trustworthiness of our prototype, we performed multiple experiments with 1,000, 10,000, and 50,000 recommendations. Figure 7.20 shows the RMSE scores of each experiment. The results demonstrate that more recommendations raise a typical error slightly. However, these results also show the effectiveness of our approach on increasingly larger samples of the dataset.

| Rating Count | RSVD | KNN-Item | KNN-User | NN-Item | NN-User |
|---|---|---|---|---|---|
| **1k** | 0.6342 | 0.4054 | 0.4239 | 0.7079 | 0.6150 |
| **10k** | 0.6817 | 0.4470 | 0.4537 | 0.7782 | 0.6352 |
| **50k** | 0.6719 | 0.4105 | 0.4281 | 0.7466 | 0.6096 |

Figure 7.20: Recommendation Accuracy of the Final Prototype Configuration

## 7.7 Conclusion

This chapter outlines our experiments and demonstrates the tuning process of our recommender system prototype. We perform focused observations that isolate various parameters over a wide range of possible values. For each experiment we analyze the results and chose the most accurate and responsive algorithm configuration. In some cases, we choose a less accurate configuration because it produced faster results at a negligible accuracy loss. Our experiments on multiple subsets of the Netflix Quiz dataset show excellent recommendation accuracy with the Pearson's Recursive approach. These results support our hypothesis that relevant data improves recommendation accuracy more than combination algorithm tuning.

# Chapter 8

# Conclusions and Future Work

The amount of online content has been continuously increasing over the past decade. As a result, users spend much more effort accessing the information they need. For example email management is a common task, where most of the messages are often irrelevant. Recommender systems were first introduced in the mail filtering context as a tool that helped users identify only the most relevant messages without reading each one. Since then, recommender systems evolved into more sophisticated algorithms that are now applied to virtually every information domain available.

The increasing popularity and growth of the World Wide Web provided the two things that make recommender systems necessary: a large catalog of content and a community of users willing to share their opinions. Therefore, now is the time to research recommender systems. The information they require is widely available and there is a great need for their services. Recommender systems are particularly useful for e-commerce applications, where customers benefit from personal shopping assistance and stores increase sales.

## 8.1 Summary

Recent research shows that it is possible to make accurate recommendations from content analysis, user behavior interpretation, and collaborative filtering techniques. We focus this research on the latter approach. However, the main obstacle in making such recommendations is the size and sparsity of modern datasets. It takes a long time to thoroughly analyze large amounts of data, so recommendation accuracy is often sacrificed to improve performance. Additionally, sparse datasets contain little usable information and establish unreliable evidence for making recommendations. Therefore, the challenge is to find a way to make accurate suggestions from sparse datasets regardless of their size.

One company that takes this challenge seriously is Netflix, a DVD rental website. In 2006, Netflix published some of its movie rating data and challenged the data mining community to produce a set of recommendations within a small error threshold. The winning algorithm would have to consistently predict actual user ratings within a 0.8572 margin on a scale from one to five. The Netflix Prize challenge became popular, and thousands of participants from major research institutions all over the world submitted their work.

We attempt to produce accurate recommendations on the Netflix dataset, one of the biggest and most sparse datasets available. This particular dataset is notoriously difficult, as many approaches fail to consistently predict its data within a small margin of error. However, the domain is very common and almost everyone can relate to and appreciate the quality of movie recommendations. Finally, major recommendation accuracy improvements have a real-life application in the Netflix system, so thousands

of customers may enjoy a more personalized DVD renting experience.

We started this research in the winter of 2008 before a significant improvement on the Netflix dataset was achieved. Previously, careful tuning of various statistics, machine learning, and information retrieval techniques provided the initial progress in the Netflix Prize challenge. On July 26th, 2009, the team named "BellKor's Pragmatic Chaos" reached the lowest error value of 0.8567.

We researched many successful solutions like this one and attempted to improve them. However, unlike existing approaches that focused on clever ways of combining input ratings, we looked for ways to select a better input. In August 2010, our prototype successfully predicted a set of 50,000 randomly chosen ratings with a typical error of 0.4105. Our method is considerably slower, but it offers better accuracy than the currently leading approach. We document our path to a low recommendation error in this dissertation.

## 8.2 Conclusion

Before we propose a new way to make accurate recommendations, we research the background and motivation for recommender systems. They fill an important niche in our everyday lives, much of which are spent online. Instead of manually inspecting gigabytes of online content, we often rely on recommender systems to simplify the information overload problem and act as an advisor, suggesting only the most relevant data. We trace the evolution of recommender systems from simple aggregation functions to personalized suggestions and identify the ways this may be achieved, be

it through explicit recommendations by your peers, information inferred from your own behavior, or item content analysis.

We focus our research on collaborative filtering recommendations because they most closely resemble the natural "word-of-mouth" suggestions. This approach applies to any kind of data and has milder limitations. In fact, all disadvantages of the collaborative filtering approach are related to dataset sparsity, which forms the basis of our research. Our goal is to develop an accurate collaborative filtering solution for sparse datasets that may not be supplemented with external data, implicit user feedback, or content analysis. This is one of the most common and most difficult problems in recommender system research today.

In order to produce recommendations from arbitrarily large datasets, we propose an architecture that can asynchronously generate recommender input. It consists of a single input generation and multiple combination algorithms. Our prototype can generate inputs once and use them multiple times, thus addressing the performance problem of memory-based collaborative filtering. Additionally, because we have a dedicated component for generating recommender input, our prototype can accommodate many datasets with various structure. Our architecture provides improved performance, modifiability, and scalability. However, it may not support certain resource-intensive combination algorithms.

Such limitation is acceptable because the accuracy of our system comes from the input data and not the combination algorithm. We research available collaborative filtering approaches and implement three popular methods: Robust Singular Value Decomposition, K Nearest Neighbors, and Neural Network. We choose these algo-

rithms because they are simple and fast. Since these algorithms reside on a client side, which is within a browser application that has limited processing abilities, speed and simplicity are crucial.

We inspect the accuracy of different recommender configurations and choose the best set of parameters for our experiments. Once tuned recommender algorithms are in place, we perform initial measurements on standard input matrices, i.e., input generated according to global similarity, the way other researchers have done it. Unfortunately, even the best approach is typically within 0.9574 of an actual rating. This supports our hypothesis that algorithm tuning has little to do with recommendation accuracy. However, our hypothesis also states that good input data can produce accurate suggestions, regardless of the algorithm used.

We believe that a small number of relevant ratings is sufficient to make an accurate recommendation. Such ratings may be chosen with local similarity, instead of a more traditional global similarity. This metric requires two neighbors to be similar in some, but not all domains. It relaxes similarity constraints, so more data becomes available. It also establishes more pertinent evidence for vector similarity, so that selected ratings are more relevant. To test this claim we develop an algorithm that organizes ratings in matrices sorted by user/item similarities.

The main purpose of our input generation approach is to produce small and dense input matrices. It selects relevant vectors according to various similarity measures. One of them relies on relative ratings or opinions normalized by that vector's mean rating. The experiments show that relative ratings produce better results because similar vectors contain matching opinions more often. We also experiment with re-

cursively resorting the input matrix, since local similarities of users and items are mutually dependent. The experiments show that just two rounds of similarity computations produce the best results. This process of generating the matrix is slow, because a large number of ratings must be retrieved, compared, and sorted multiple times. However, the resulting recommendation accuracy justifies the performance drawback.

The benefits of local similarity are the basis for our input generation algorithm. To determine the effectiveness of our hypothesis, we perform an empirical evaluation of three combination algorithms paired with our input generation component. The results show that tuning recommender algorithms has little effect on recommendation accuracy. However, all algorithms produce better results when supplied with more relevant input. Our prototype shows excellent recommendation accuracy on a number of random samples from the Netflix dataset. These results support our hypothesis that relevant data improves recommendation accuracy more than combination algorithm tuning.

## 8.3 Limitations of the Study

Recommender systems can model as well as shape user preferences. People often value recommended items more than they would otherwise, simply because items were suggested. Sundar, Oeldorf-Hirsch, and Xu refer to this phenomenon as the bandwagon effect, e.g., "if others think that something is good, then I should, too" [154, 155]. Furthermore, Cosley et al. empirically demonstrate that user ratings are biased in

the direction of presented recommendation [41]. The authors discover that users tend to rate toward a recommendation whether it is accurate or not, e.g., rating a recommended movie as four stars, even though you would normally give it three stars. In fact, users have an inherent trust towards recommender systems [63]. We focus on the accuracy of recommendations, regardless of the content or intended purpose of suggestions. However, future research into the use and abuse [88, 135, 137] of the recommender system influence would be very interesting.

User privacy is also often abused. Privacy is especially relevant for domains where users express opinions on controversial topics, e.g., politics, religion, and relationships. Moreover, people also have the right to prevent their less passionate opinions from becoming widely known. In November 2010, Google settled a $8.5 million lawsuit against its news recommender system. This system helped Gmail users share updates, pictures, and videos with other Gmail accounts. However, sharing preferences automatically included most often used contacts without the owner's explicit permission. This feature was meant to enhance user experience but instead threatened their privacy. As a result, ignoring privacy considerations turned out to be a costly mistake.

To protect themselves, some recommender systems employ user pseudonyms or cryptographic techniques that encrypt data [20, 99, 131]. However, even when these provisions are in place, it is possible to uniquely identify different users simply by capturing a few of their votes [111]. This issue has motivated further research on improving privacy in existing recommender systems [30, 31, 85, 105, 128, 172]. This particular project does not involve privacy considerations, because Netflix protects

users' identities with arbitrary numeric IDs within its dataset.

Recommender systems need to know users' preferences to operate. This information is not abundantly available, but there are always customers willing to share their explicit opinion, e.g., rating a restaurant on a five-star scale. Some recommender systems also use implicit user opinions, or data inferred from user behavior, e.g., the most often visited restaurant is probably a favorite [26]. These estimates can reinforce the recommendation model, which leads to better suggestions [12,63]. However, using implicit feedback invades user privacy since the users did not explicitly provide that information. Our prototype only considers explicit user opinions, but future research into making effective conclusions from both types of user feedback would be interesting.

Although our research is built around predicting explicit opinions, our work could also be beneficial for Top-N recommenders, or systems that estimate a ranking of items. The focus of such systems is not to recommend the best possible item, but to predict a set of items that fit well together [44, 75, 83, 142]. For example, omitting a good song from a music playlist has little effect on user experience, yet recommending a list of great songs can be considered a poor recommendation if they do not complement each other. With systems like Pandora Radio gaining popularity, Top-N recommender research is interesting, but we focus on making individual recommendations because they have more applications across a wider range of domains.

The quality of individual recommendations is often characterized by the prediction error, yet reducing this error has little effect on improving the overall quality of the system. Observing and surveying the users of a system would be a much better way

to evaluate its quality because a set of completely useless recommendations may be accurate [65]. For example, paper towels and trash bags are popular items in any grocery store because customers often need them, and a recommender system that suggests these obvious items does not provide any value to the customers. Since we have no access to the users, we are limited to offline analysis of the prediction error, i.e., we can only verify the presence of existing preferences [99]. However, considering the popularity of Netflix, we could put together a test group of actual users in the future.

## 8.4 Future Work

We plan to extend our research in two distinct directions, serendipitous recommendations and recommendation explanations. These two attributes contribute to customer satisfaction as much as recommendation accuracy. However, unlike recommendation accuracy, there are less limitations to how effective they can be. These research problems may be difficult to solve, but they would be instantly recognized and appreciated by the users. In fact, serendipitous recommendations and recommendation explanations are both recurring topics at the ACM Conference on Recommender Systems.

Ability to make serendipitous recommendations is one of the reasons we focused our research on the collaborative filtering recommender systems. Recommendation novelty and serendipity directly affect user satisfaction [63]. Therefore, in addition to calculating the likelihood that a user will enjoy an item, a good recommender system will suggest items that are novel and serendipitous. Serendipitous recommendations

allow a user to appreciate the content that they might not have otherwise discovered [17, 63]. However, the challenge with making such recommendations is that by definition serendipitous suggestions have no evidence to support their existence.

Accurate and surprisingly interesting recommendations may be entertaining to a casual user, but they may not inspire trust from a highly invested user. Recommendation explanations clarify suggestions and build the user's trust with the system [62]. Both of these qualities positively affect user satisfaction. This is particularly evident in recommender systems where the cost of a bad recommendation is significant, e.g., purchasing a worthless piece of real estate as opposed to renting a boring movie. In fact, serendipitous recommendations improve user satisfaction only if the cost of a bad suggestion is negligible. However, as the user's investment becomes more substantial, the novelty of the recommended item becomes secondary to the recommendation accuracy [63]. Therefore, any future research must balance the recommendation accuracy and novelty in order to pleasantly surprise users with useful and accurate recommendations.

When receiving serendipitous recommendations, a user may be curious as to why the recommender system would make these suggestions. Recommendation explanations reveal such reasoning, which may not be obvious. The user can then judge the trustworthiness of a suggestion and make an informed decision. For example, if there is not enough information about the active user and the recommendation is based on a global average, he/she should be made aware of that fact. This way, the user is informed and not discouraged when the first few recommendations turn out to be poor [13, 63]. Our future research will help recommender systems justify their sugges-

tions so that users are satisfied and aware of the reasons why such recommendations were made.

Finally, we would like to improve the performance of our recommender system design. A major disadvantage of our current prototype is the recommendation speed of about 400 suggestions per hour. Fortunately, the architecture allows us to preprocess the input data, thus making the user interface more responsive. However, we would like to generate the same recommendations interactively without sacrificing the quality. To achieve this, we plan to research cloud computing, clustered database implementations, and different hardware configurations that may improve recommendation performance.

In particular, Amazon Elastic Compute Cloud Web service provides a cost effective way to interactively produce recommendations with our prototype. For instance, High-CPU or Cluster Compute instances provide vast processing ability with increased network performance. They are well suited for high performance computing applications like ours. Furthermore, the auto scaling feature automatically adjusts the number of computing instances depending on the amount of incoming requests and the elastic load balancing feature automatically distributes incoming traffic across multiple computing instances. Amazon's service is ideal for our future prototype implementation because it provides high performance along with the elasticity, flexibility, and cost advantages.

Recommender system research is a relatively new field and there are plenty of unresolved problems within it. Nevertheless, it already encompasses a large amount of research on a variety of recommendation approaches. We focus on collaborative

filtering recommendations and outline some of our most prominent research aspirations. However, applying recommender systems to more abstract domains could lead to even more interesting questions. For instance, a healthcare organization, called Heritage Provider Network, recently announced a $3 million challenge to predict the likelihood of an individual's future hospitalization. A successful solution could improve patient lives and support the economy. Therefore, our present and future work is relevant and applicable to many non-trivial applications. –

# Appendix: Source Code

# Appendix A

# Input Generation Procedure

The input generation procedure requests the next available user/item tuple, produces an input matrix for it, and marks it as done. It uses *get_next_tuple* procedure to reserve a tuple and then calls the necessary procedures. It first creates the temporary database schema, then loads the relevant ratings, sorts the matrix, truncates it, and saves it. This process may also mark all tuples as available with the *reset_flag* option.

Multiple instances of this procedure may execute simultaneously, because the permanent database schema is used for reading only. There are no time-consuming writes, so multiple instances may search the dataset at the same time. Likewise, simultaneous processes write to temporary schema instances that exists only within their respective sessions, so there is no competition for the write access to a shared resource. The only such resource is the *tbl_matrices* table, which gets accessed with infrequent (every 1-3 seconds), lightweight (single digit) updates of the matrix status. Any possible delay due to a read/write lockout on this table is negligible relative to the overall component latency.

Listing A.1: Input Generation Procedure – make_matrices.sh

```
#!/bin/bash

# make matrices for all user-item tuples in tbl_matrices
# session field stores the state of a matrix making process
# matrix states: 0 - no matrix, 1 - in progress, 2 - done
# usage ./make_matrices.sh reset_flag

DB_NAME="morozosl"
DONE_FLAG=0

if [ "$1" = "reset" ]
then
    mysql $DB_NAME -e "UPDATE tbl_matrices SET session=0"
fi

while [ $DONE_FLAG -eq 0 ]
do
    # ask for the next user-item tuple
    output=`mysql $DB_NAME -Ne "SELECT get_next_tuple()"`

    # set internal field separator and separate user-item tuple
    IFS="_"
    declare -a tuple=($output)

    if [ `echo "${tuple[0]}" | grep "^[0-9]\+$"` ]
    then
        if [ `echo ${tuple[1]} | grep "^[0-9]\+$"` ]
        then
            # make one matrix with user_id = ${tuple[0]}, item_id = ${tuple[1]}
            sql="
            # enable large (5GB) memory tables
            SET max_heap_table_size = 5368709120;
            SET tmp_table_size      = 5368709120;

            # set session variables
            SET @active_user_id = ${tuple[0]};
            SET @active_item_id = ${tuple[1]};
            SET @matrix_size    = 30;

            CALL create_tmp_tables();
            CALL load_matrix();
            CALL sort_matrix();
            CALL truncate_matrix();
            CALL save_matrix();
            "
            #echo $sql

            mysql $DB_NAME -Ne "$sql"
        else
            DONE_FLAG=1
        fi
    else
        DONE_FLAG=1
    fi
done
```

# Appendix B

# Matrix Reservation Function

The matrix reservation procedure selects a single unprocessed user/item tuple and marks it with the connection ID. Every database connection has an ID that is unique among currently connected clients. Therefore, there should be only one tuple with such a mark. The function then saves the selected user and item IDs, removes the mark to ensure next tuple marked with this connection ID will be unique, and returns the reserved user/item IDs as an underscore-separated string. We use session variables so that this function may be executed manually. This way, we can "step through" the matrix generation process for debugging purposes.

Listing B.1: Matrix Reservation Function – get_next_tuple.sql

```
FUNCTION get_next_tuple() RETURNS varchar(17)
BEGIN

# matrix states: 0 - no matrix, 1 - in progress, 2 - done

# mark one available tuple with this connection id
UPDATE      tbl_matrices
SET         session=CONNECTION_ID()
WHERE       session=0
LIMIT       1;


# get the user and item ids
SELECT      user_id, item_id
INTO        @active_user_id, @active_item_id
FROM        tbl_matrices
WHERE       session=CONNECTION_ID();


# mark the tuple as reserved
UPDATE      tbl_matrices
SET         session=1
WHERE       session=CONNECTION_ID();

RETURN      CONCAT(@active_user_id, '_', @active_item_id);

END
```

# Appendix C

# Matrix Setup Procedure

The setup process creates a temporary database schema. The schema consists of seven relations, three user-related, three item-related, and one matrix relation. The user-related relations use *user_id* as the primary key. Likewise, item-related relations rely on *item_id* as a way to uniquely identify records. The matrix relation does not need a primary key because we never search for a particular rating. Instead, it contains two indices, because we usually search for a set of ratings that belong to a particular user or item vector.

We store these tables in memory, because this storage engine provides the fastest data access. We also declare the tables as temporary, which means they exist only within the context of the active database session. In fact, multiple database sessions running this procedure will create individual copies of the temporary schema. As a result, each matrix generation process can refer to different data by the same name. This practice enforces naming consistency and reduces the complexity of our recommender system.

Listing C.1: Matrix Setup Procedure – setup_matrix.sql

```
1  PROCEDURE create_tmp_tables()
2  BEGIN
3
4  # similarity and product could be negative
5
6  CREATE TEMPORARY TABLE tmp_items (
7    item_id mediumint(8) unsigned NOT NULL,
8    rating tinyint(1) unsigned NOT NULL,
9    similarity decimal(4,3) NOT NULL default '0.000',
10   PRIMARY KEY  (item_id)
11 ) ENGINE=MEMORY DEFAULT CHARSET=latin1;
12
13 CREATE TEMPORARY TABLE tmp_items_best (
14   item_id mediumint(8) unsigned NOT NULL,
15   PRIMARY KEY  (item_id)
16 ) ENGINE=MEMORY DEFAULT CHARSET=latin1;
17
18 CREATE TEMPORARY TABLE tmp_items_calc (
19   item_id mediumint(8) unsigned NOT NULL,
20   length decimal(8,3) unsigned NOT NULL,
21   product decimal(8,3) NOT NULL,
22   PRIMARY KEY  (item_id)
23 ) ENGINE=MEMORY DEFAULT CHARSET=latin1;
24
25 CREATE TEMPORARY TABLE tmp_matrix (
26   user_id mediumint(8) unsigned NOT NULL,
27   item_id mediumint(8) unsigned NOT NULL,
28   rating tinyint(1) unsigned NOT NULL,
29   KEY user_id (user_id),
30   KEY item_id (item_id)
31 ) ENGINE=MEMORY DEFAULT CHARSET=latin1;
32
33 CREATE TEMPORARY TABLE tmp_users (
34   user_id mediumint(8) unsigned NOT NULL,
35   rating tinyint(1) unsigned NOT NULL,
36   similarity decimal(4,3) NOT NULL default '0.000',
37   PRIMARY KEY  (user_id)
38 ) ENGINE=MEMORY DEFAULT CHARSET=latin1;
39
40 CREATE TEMPORARY TABLE tmp_users_best (
41   user_id mediumint(8) unsigned NOT NULL,
42   PRIMARY KEY  (user_id)
43 ) ENGINE=MEMORY DEFAULT CHARSET=latin1;
44
45 CREATE TEMPORARY TABLE tmp_users_calc (
46   user_id mediumint(8) unsigned NOT NULL,
47   length decimal(8,3) unsigned NOT NULL,
48   product decimal(8,3) NOT NULL,
49   PRIMARY KEY  (user_id)
50 ) ENGINE=MEMORY DEFAULT CHARSET=latin1;
51
52 END
```

# Appendix D

# Load Matrix Procedure

Load matrix procedure creates the initial matrix with all potentially relevant ratings. First, it determines a list of all users who rated the active item in *tmp_users* and a list of all items rated by the active user in *tmp_items*. These lists also form the initial best dimensions of the matrix. Finally, this process populates *tmp_matrix* with all ratings associated with relevant users in *tmp_users* and relevant items in *tmp_items*.

Note that *tmp_items* table represents the active user vector and *tmp_users* table represents the active item vector. We ensure that the active user is part of the active item vector, but replace its rating, if there is one. The same applies to the active user vector. The two active vectors form a point of reference for the sorting procedure. Note that both vectors contain the active rating, but because it is zero, it does not participate in similarity computations. Therefore, even if *tmp_matrix* contains the active rating, it does not affect the contents of the matrix.

Listing D.1: Load Matrix Procedure – load_matrix.sql

```
PROCEDURE load_matrix()
BEGIN

# populate tmp_users and tmp_items
TRUNCATE    tmp_users;

INSERT INTO tmp_users(user_id, rating)
SELECT      user_id, rating
FROM        tbl_ratings
WHERE       item_id=@active_item_id;

TRUNCATE    tmp_items;

INSERT INTO tmp_items(item_id, rating)
SELECT      item_id, rating
FROM        tbl_ratings
WHERE       user_id=@active_user_id;


# guarantee a record for active user and active item
REPLACE INTO tmp_users(user_id, rating) VALUES(@active_user_id, 0);
REPLACE INTO tmp_items(item_id, rating) VALUES(@active_item_id, 0);


# initially all vectors are considered best
TRUNCATE    tmp_users_best;

INSERT INTO tmp_users_best(user_id)
SELECT      user_id
FROM        tmp_users;

TRUNCATE    tmp_items_best;

INSERT INTO tmp_items_best(item_id)
SELECT      item_id
FROM        tmp_items;


# populate tmp_matrix with ALL relevant ratings
TRUNCATE    tmp_matrix;

# smaller table (items) should go first
INSERT INTO tmp_matrix(user_id, item_id, rating)
SELECT      users.user_id, items.item_id, ratings.rating
FROM        tbl_ratings ratings
JOIN        tmp_items items ON ratings.item_id=items.item_id
JOIN        tmp_users users ON ratings.user_id=users.user_id;

END
```

# Appendix E

# Sort Matrix Procedure

The sort procedure computes vector similarities and keeps track of the best vectors in dedicated tables. It calls two sub-procedures, *compute_similarities* and *save_similarities*, to perform this task. Furthermore, this procedure determines if resorting is necessary for this particular matrix shape. We precompute users' and items' rating counts to improve performance. Sorting and resorting may not be required for small matrices. In such cases, this procedure may choose to skip one or both sorting cycles.

Listing E.1: Sort Matrix Procedure – sort_matrix.sql

```
1  PROCEDURE sort_matrix()
2  BEGIN
3
4  DECLARE     relevant_item_count MEDIUMINT(8);
5  DECLARE     relevant_user_count MEDIUMINT(8);
6
7  # save rating counts for active vectos
8  SELECT      rating_count
9  INTO        relevant_item_count
10 FROM        tbl_users
11 WHERE       user_id=@active_user_id;
12
13 SELECT      rating_count
14 INTO        relevant_user_count
15 FROM        tbl_items
16 WHERE       item_id=@active_item_id;
17
18
19 IF (relevant_item_count > @matrix_size OR relevant_user_count > @matrix_size) THEN
20
21     # sort the matrix for the first time (global similarity)
22     CALL compute_similarities();
23     CALL save_similarities();
24
25     IF (relevant_item_count > @matrix_size AND relevant_user_count > @matrix_size)
       THEN
26
27         # sort the matrix one more time (local similarity)
28         CALL compute_similarities();
29         CALL save_similarities();
30
31     END IF;
32
33 END IF;
34
35 END
```

# Appendix F

# Truncate Matrix Procedure

Truncate procedure reduces the matrix to its final dimensions. It removes all users and items that are not listed in *tmp_users_best* and *tmp_items_best* respectively. The resulting matrix is considerably smaller and consists of the most relevant data. Note that it is faster to truncate the matrix and repopulate it then to delete irrelevant ratings that make up most of the matrix. However, matrix truncation is only necessary for serialization. In some production systems, truncation could be omitted in favor of performance. For instance, the KNN approach could generate recommendations as a weighted average of the ratings in *tmp_users* or *tmp_items*. This method does not require a matrix, but it does need vector similarities. Therefore, some recommender systems may discard the matrix as soon as it has been sorted.

Listing F.1: Truncate Matrix Procedure – truncate_matrix.sql

```
1  PROCEDURE truncate_matrix()
2  BEGIN
3
4  # delete all but the best users
5  DELETE      tmp_users
6  FROM        tmp_users
7  LEFT JOIN   tmp_users_best
8  ON          tmp_users.user_id=tmp_users_best.user_id
9  WHERE       tmp_users_best.user_id IS NULL;
10
11 # delete all but the best items
12 DELETE      tmp_items
13 FROM        tmp_items
14 LEFT JOIN   tmp_items_best
15 ON          tmp_items.item_id=tmp_items_best.item_id
16 WHERE       tmp_items_best.item_id IS NULL;
17
18
19 # repopulate tmp_matrix, with BEST relevant ratings
20 # this is faster than deleting non-relevant rows
21 TRUNCATE    tmp_matrix;
22
23 # smaller table (items) should go first
24 INSERT INTO tmp_matrix(user_id, item_id, rating)
25 SELECT      users.user_id, items.item_id, ratings.rating
26 FROM        tbl_ratings ratings
27 JOIN        tmp_items_best items ON ratings.item_id=items.item_id
28 JOIN        tmp_users_best users ON ratings.user_id=users.user_id;
29
30 END
```

# Appendix G

# Save Matrix Procedure

Save matrix procedure accomplishes two things, it marks the matrix as done in *tbl_matrices* and saves its contents as a newline-delimited set of item ratings, where missing ratings are replaced with a zero. Note that row/column sorting is not necessary, but it does help us verify that the active user and item vectors represent the first column and row in a serialized matrix. In a production system, caching the matrix may not be necessary, as recommendations may be generated from *tmp_matrix* directly. However, for the purposes of our empirical study, precomputed and serialized input matrices prevent redundant processing.

Listing G.1: Save Matrix Procedure – save_matrix.sql

```
PROCEDURE save_matrix()
BEGIN

# mark the matrix as done and save its contents
# matrix states: 0 - no matrix, 1 - in progress, 2 - done
# record ITEM vectors in order of decreasing similarity
# replace missing ratings with 0

UPDATE  tbl_matrices
SET     session=2,
        matrix=
        (
        SELECT      GROUP_CONCAT(line SEPARATOR "\n")
        FROM        (
                    SELECT      GROUP_CONCAT(
                                IF(matrix.rating IS NULL, 0, matrix.rating)
                                ORDER BY users.similarity DESC
                                SEPARATOR ''
                                ) as line
                    FROM        tmp_users as users
                    JOIN        tmp_items as items
                    LEFT JOIN   tmp_matrix as matrix
                    ON          users.user_id=matrix.user_id
                    AND         items.item_id=matrix.item_id
                    GROUP BY    items.item_id
                    ORDER BY    items.similarity DESC
                    ) as a
        )
WHERE   user_id=@active_user_id
AND     item_id=@active_item_id;

END
```

# Appendix H

# Cosine Similarity Procedure

The cosine similarity implementation of the *compute_similarities* procedure computes the dot product and vector length of every neighbor and the active vector. For every user vector, we take each rating and multiply it by a corresponding rating on the same item by the active user. The sum of such products constitutes the dot product of the two vectors. This quantity may then be divided by the two vector magnitudes to arrive at the cosine of the angle between them.

Note that this procedure only computes the two components of the cosine similarity, but not the measure itself. We do so because Pearson's correlation uses a very similar structure. To avoid redundancy, we separate the common functionality into a separate procedure, *save_similarities*. Also, vector comparison is restricted to the best common dimensions. For example, two users are compared only across their opinions on items listed in *tmp_items_best*. Likewise, items are compared only across the best users.

Listing H.1: Cosine Similarity Procedure – compute_similarities.sql

```
PROCEDURE compute_similarities()
BEGIN

# compute length and product from raw ratings
# we can not update tmp_users and tmp_items directly
# because update statements can not have a "group by" clause

TRUNCATE    tmp_users_calc;

INSERT INTO tmp_users_calc(user_id, length, product)
SELECT      b.user_id,
            SQRT(SUM(POW(b.rating, 2))),
            SUM(a.rating*b.rating)
FROM        tmp_items as a
JOIN        tmp_items_best as best ON a.item_id=best.item_id
JOIN        tmp_matrix as b ON a.item_id=b.item_id
GROUP BY    b.user_id;

TRUNCATE    tmp_items_calc;

INSERT INTO tmp_items_calc(item_id, length, product)
SELECT      b.item_id,
            SQRT(SUM(POW(b.rating, 2))),
            SUM(a.rating*b.rating)
FROM        tmp_users as a
JOIN        tmp_users_best as best ON a.user_id=best.user_id
JOIN        tmp_matrix as b ON a.user_id=b.user_id
GROUP BY    b.item_id;

END
```

# Appendix I

# Pearson's Correlation Procedure

The Pearson's correlation implementation of the *compute_similarities* procedure also computes the dot product and vector length of every neighbor and the active vector. It is identical to cosine similarity in every way, except the ratings are normalized by the vector average. Instead of using raw ratings, this procedure uses their deviations from the average rating. We precompute users' and items' average ratings to improve performance.

Note that even though original ratings are positive, their deviations from the mean may be negative. For example, a person rating a movie as three stars, when they typically rate movies as four stars, will result in a $3 - 4 = -1$ normalized rating. Negative ratings may produce negative similarities, which is why we model the similarity field as a signed decimal in *tmp_users* and *tmp_items* (See Appendix: Listing C)

Listing I.1: Pearson's Correlation Procedure – compute_similarities.sql

```
PROCEDURE compute_similarities()
BEGIN

DECLARE     active_user_avg_rating DECIMAL(4,3);
DECLARE     active_item_avg_rating DECIMAL(4,3);

# save average ratings for active vectos
SELECT      avg_rating
INTO        active_user_avg_rating
FROM        tbl_users
WHERE       user_id=@active_user_id;

SELECT      avg_rating
INTO        active_item_avg_rating
FROM        tbl_items
WHERE       item_id=@active_item_id;


# compute length and product from normalized ratings
# we can not update tmp_users and tmp_items directly
# because update statements can not have a "group by" clause

TRUNCATE    tmp_users_calc;

INSERT INTO tmp_users_calc(user_id, length, product)
SELECT      b.user_id,
            SQRT(SUM(POW(b.rating-c.avg_rating, 2))),
            SUM((a.rating-active_user_avg_rating)*(b.rating-c.avg_rating))
FROM        tmp_items as a
JOIN        tmp_items_best as best ON a.item_id=best.item_id
JOIN        tmp_matrix as b ON a.item_id=b.item_id
JOIN        tbl_users as c ON b.user_id=c.user_id
WHERE       a.rating > 0
GROUP BY    b.user_id;

TRUNCATE    tmp_items_calc;

INSERT INTO tmp_items_calc(item_id, length, product)
SELECT      b.item_id,
            SQRT(SUM(POW(b.rating-c.avg_rating, 2))),
            SUM((a.rating-active_item_avg_rating)*(b.rating-c.avg_rating))
FROM        tmp_users as a
JOIN        tmp_users_best as best ON a.user_id=best.user_id
JOIN        tmp_matrix as b ON a.user_id=b.user_id
JOIN        tbl_items as c ON b.item_id=c.item_id
WHERE       a.rating > 0
GROUP BY    b.item_id;

END
```

# Appendix J

# Save Similarities Procedure

The save similarities procedure completes the similarity computation process and saves all item and user similarities. It also ensures that the two active vectors receive the highest possible similarities. In other words, the active item vector should be considered most similar to itself. The same requirement apples to the active user similarity. This adjustment is necessary to neutralize the rounding error which may cause other, very similar, vectors to appear as the first row/column of the serialized matrix. Finally, this procedure records the vectors with highest similarities as best matrix dimensions. Note that the size of these lists is restricted by the *matrix_size* session variable and they contain vectors with positive similarities only.

Listing J.1: Save Similarities Procedure – save_similarities.sql

```
PROCEDURE save_similarities()
BEGIN

DECLARE     active_user_length DECIMAL(8,3);
DECLARE     active_item_length DECIMAL(8,3);

# save active vector lengths
SELECT      length
INTO        active_user_length
FROM        tmp_users_calc
WHERE       user_id=@active_user_id;

SELECT      length
INTO        active_item_length
FROM        tmp_items_calc
WHERE       item_id=@active_item_id;


# compute similarities with every vector
UPDATE      tmp_users as a, tmp_users_calc as b
SET         a.similarity=b.product/(active_user_length*b.length)
WHERE       a.user_id=b.user_id;

UPDATE      tmp_items as a, tmp_items_calc as b
SET         a.similarity=b.product/(active_item_length*b.length)
WHERE       a.item_id=b.item_id;


# active vectors should always be most similar
UPDATE      tmp_users
SET         similarity=9.999
WHERE       user_id=@active_user_id;

UPDATE      tmp_items
SET         similarity=9.999
WHERE       item_id=@active_item_id;


# make a list of @matrix_size best vectors
# EXECUTE command only works with session variables
PREPARE     record_best_users
FROM        "INSERT INTO tmp_users_best(user_id)
            SELECT user_id FROM tmp_users WHERE similarity>0
            ORDER BY similarity DESC LIMIT ?";

PREPARE     record_best_items
FROM        "INSERT INTO tmp_items_best(item_id)
            SELECT item_id FROM tmp_items WHERE similarity>0
            ORDER BY similarity DESC LIMIT ?";

TRUNCATE    tmp_users_best;
EXECUTE     record_best_users USING @matrix_size;

TRUNCATE    tmp_items_best;
EXECUTE     record_best_items USING @matrix_size;

DEALLOCATE PREPARE record_best_users;
DEALLOCATE PREPARE record_best_items;

END
```

# Bibliography

[1] 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 Knowledge and Data Engineering*, 17(6):734–749, 2005.

[2] Charu C. Aggarwal, Joel L. Wolf, Kun-Lung Wu, and Philip S. Yu. Horting hatches an egg: A new graph-theoretic approach to collaborative filtering. In *Proceedings of the Fifth ACM SigKDD International Conference on Knowledge Discovery and Data Mining*, pages 201–212, New York, NY, USA, 1999. ACM.

[3] Adam LaPitz George Karypis Al Mamunur Rashid, Shyong K. Lam and John Riedl. Towards a scalable kNN CF algorithm: Exploring effective applications of clustering. In *Web Mining and Web Usage Analysis*, 2008.

[4] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 1997.

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

[6] Linas Baltrunas and Francesco Ricci. Locally adaptive neighborhood selection for collaborative filtering recommendations. In *AH '08: Proceedings of the 5th international conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, pages 22–31, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Linas Baltrunas and Francesco Ricci. *Knowledge Discovery Enhanced with Semantic and Social Information*, volume 220/2009. Springer Berlin / Heidelberg, 2009.

[8] Shumeet Baluja, Rohan Seth, D. Sivakumar, Yushi Jing, Jay Yagnik, Shankar Kumar, Deepak Ravichandran, and Mohamed Aly. Video suggestion and discovery for youtube: Taking random walks through the view graph. In *Proceedings of the 17th International Conference on World Wide Web*, pages 895–904, New York, NY, USA, 2008. ACM.

[9] Justin Basilico and Thomas Hofmann. Unifying collaborative and content-based filtering. In *Proceedings of the Twenty-First International Conference on Machine Learning*, page 9, New York, NY, USA, 2004. ACM.

[10] Chumki Basu, Haym Hirsh, and William Cohen. Recommendation as classification: Using social and content-based information in recommendation. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 714–720, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[11] Robert Bell, Yehuda Koren, and Chris Volinsky. Modeling relationships at multiple scales to improve accuracy of large recommender systems. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 95–104, New York, NY, USA, 2007. ACM.

[12] Robert M. Bell and Yehuda Koren. Lessons from the netflix prize challenge. *ACM SIGKDD Explorations Newsletter*, 9(2):75–79, 2007.

[13] Robert M. Bell and Yehuda Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *Proceedings of the Seventh IEEE International Conference on Data Mining*, pages 43–52. IEEE Computer Society, 2007.

[14] Robert M. Bell, Yehuda Koren, and Chris Volinsky. The bellkor solution to the netflix prize. Technical report, AT&T Labs - Research, 2007.

[15] James Bennett, Charles Elkan, Bing Liu, Padhraic Smyth, and Domonkos Tikk. KDD cup and workshop 2007. *ACM SIGKDD Explorations Newsletter*, 9(2):51–52, 2007.

[16] James Bennett and Lanning Stan. The netflix prize. In *KDD Cup and Workshop 2007*, 2007.

[17] Jim Bennett. The Cinematch system: Operation, scale coverage, accuracy, impact. Summer School on the Present and Future of Recommender Systems, 2006.

[18] Joel Bennett. A collaborative filtering recommender using SOM clustering on keywords. Master's thesis, Rochester Institute of Technology, 2006.

[19] Shlomo Berkovsky. Decentralized mediation of user models for a better personalization. In *Adaptive Hypermedia and Adaptive Web-Based Systems, 4th International Conference, AH 2006, Dublin, Ireland, June 21-23, 2006, Proceedings*, pages 404–408, 2006.

[20] Shlomo Berkovsky, Yaniv Eytani, Tsvi Kuflik, and Francesco Ricci. Enhancing privacy and preserving accuracy of a distributed collaborative filtering. In *Proceedings of the 2007 ACM Conference on Recommender Systems*, pages 9–16, New York, NY, USA, 2007. ACM.

[21] Shlomo Berkovsky, Tsvi Kuflik, and Francesco Ricci. Cross-domain mediation in collaborative filtering. In *UM '07: Proceedings of the 11th international conference on User Modeling*, pages 355–359, Berlin, Heidelberg, 2007. Springer-Verlag.

[22] Shlomo Berkovsky, Tsvi Kuflik, and Francesco Ricci. Distributed collaborative filtering with domain specialization. In *Proceedings of the 2007 ACM Conference on Recommender Systems*, pages 33–40, New York, NY, USA, 2007. ACM.

[23] Daniel Billsus and Michael J. Pazzani. Learning collaborative information filters. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 46–54, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[24] Daniel Billsus and Michael J. Pazzani. User modeling for adaptive news access. *User Modeling and User-Adapted Interaction*, 10(2-3):147–180, 2000.

[25] David Blackwell. Conditional expectation and unbiased sequential estimation. *The Annals of Mathematical Statistics*, 18(1):105–110, 1947.

[26] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, pages 43–52, 1998.

[27] Lukas Brozovsky. ColFi - recommender system for a dating service. Master's thesis, Charles University in Prague, 2006.

[28] Robin Burke. Knowledge-based recommender systems. In *Encyclopedia of Library and Information Systems*, volume 69, 2000.

[29] L. Candillier, F. Meyer, and M. Boullé. Comparing state-of-the-art collaborative filtering systems. In *Proceedings of the 5th International Conference on Machine Learning and Data Mining in Pattern Recognition*, volume 4571 of *LNCS*, pages 548–562. Springer, 2007.

[30] John Canny. Collaborative filtering with privacy. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 45, Washington, DC, USA, 2002. IEEE Computer Society.

[31] John Canny. Collaborative filtering with privacy via factor analysis. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 238–245, New York, NY, USA, 2002. ACM.

[32] George Casella and Christian P. Robert. Rao-blackwellisation of sampling schemes. *Biometrika*, 83(1):81–94, March 1996.

[33] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM Sigcomm Conference on Internet Measurement*, pages 1–14, New York, NY, USA, 2007. ACM.

[34] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[35] Annie Chen. Context-aware collaborative filtering system: Predicting the user's preferences in ubiquitous computing. In *Extended Abstracts on Human Factors in Computing Systems*, pages 1110–1111, New York, NY, USA, 2005. ACM.

[36] Qiyang Chen. Modeling a user's domain knowledge with neural networks. *International Journal of Human-Computer Interaction*, 9(1):25–40, 1997.

[37] Jinhyung Cho, Kwiseok Kwon, and Yongtae Park. Collaborative filtering using dual information sources. *IEEE Intelligent Systems*, 22(3):30–38, 2007.

[38] Stefan Schaal Chris Atkeson, Andrew Moore. Locally weighted learning. *AI Review*, 11:11–73, April 1997.

[39] Mark Claypool, Anuja Gokhale, Tim Miranda, Pavel Murnikov, Dmitry Netes, and Matthew Sartin. Combining content-based and collaborative filters in an online newspaper. In *Proceedings of ACM SIGIR Workshop on Recommender Systems*, August 1999.

[40] M. Connor and J. Herlocker. Clustering items for collaborative filtering, 2001.

[41] Dan Cosley, Shyong K. Lam, Istvan Albert, Joseph A. Konstan, and John Riedl. Is seeing believing? how recommender interfaces affect users' opinions. *CHI Letters*, 5:585–592, 2003.

[42] I. Crnkovic. Component-based software engineering - new challenges in software development. In *Information Technology Interfaces, 2003. ITI 2003. Proceedings of the 25th International Conference on*, pages 9–18, 2003.

[43] Ayhan Demiriz. Enhancing product recommender systems on sparse binary data. *Data Mining and Knowledge Discovery*, 9:147–170, 2004. 10.1023/B:DAMI.0000031629.31935.ac.

[44] Mukund Deshpande and George Karypis. Item-based top-n recommendation algorithms. *ACM Transactions on Information Systems*, 22(1):143–177, 2004.

[45] Yi Ding and Xue Li. Time weight collaborative filtering. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, pages 485–492, New York, NY, USA, 2005. ACM.

[46] Yi Ding, Xue Li, and Maria E. Orlowska. Recency-based collaborative filtering. In *Proceedings of the 17th Australasian Database Conference*, pages 99–107, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[47] David Edmond. Refining database systems. In *Proceedings of the 9th International Conference of Z Usres on The Z Formal Specification Notation*, pages 25–44, London, UK, 1995. Springer-Verlag.

[48] Charles Elkan. Rao-blackwell theorem: Rao-blackwell theorem: Intuition, lemmas and start of proof, 2005.

[49] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. Chair-Taylor, Richard N.

[50] Dan Frankowski, Shyong K. Lam, Shilad Sen, F. Maxwell Harper, Scott Yilek, Michael Cassano, and John Riedl. Recommenders everywhere: The WikiLens community-maintained recommender system. In *Proceedings of the 2007 International Symposium on Wikis*, pages 47–60, New York, NY, USA, 2007. ACM.

[51] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse is so Hard? *IEEE Software*, 12(6):17–26, 1995.

[52] David Garlan. Software architecture: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, New York, NY, USA, 2000. ACM.

[53] David Garlan, Robert Monroe, and David Wile. Acme: An architecture description interchange language. In *in Proceedings of CASCON'97*, pages 169–183, 1997.

[54] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[55] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[56] Nathaniel Good, J. Ben Schafer, Joseph A. Konstan, Al Borchers, Badrul Sarwar, Jon Herlocker, and John Riedl. Combining collaborative filtering with personal agents for better recommendations. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 439–446. AAAI Press, 1999.

[57] Vu Ha and Peter Haddawy. Toward case-based preference elicitation: Similarity measures on preference structures. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 193–201, 1998.

[58] Eui-Hong (Sam) Han and George Karypis. Feature-based recommendation system. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, pages 446–452, New York, NY, USA, 2005. ACM.

[59] David Heckerman, David Maxwell Chickering, Christopher Meek, Robert Rounthwaite, and Carl Kadie. Dependency networks for inference, collaborative filtering, and data visualization. *Journal of Machine Learning Research*, 1:49–75, 2001.

[60] Joseph M. Hellerstein, Michael Stonebraker, and James R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[61] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 230–237, New York, NY, USA, 1999. ACM.

[62] Jonathan L. Herlocker, Joseph A. Konstan, and John Riedl. Explaining collaborative filtering recommendations. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, pages 241–250, New York, NY, USA, 2000. ACM.

[63] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems*, 22(1):5–53, 2004.

[64] Jonathan Lee Herlocker. *Understanding and Improving Automated Collaborative Filtering Systems*. PhD thesis, University of Minnesota, 2000. Adviser-Joseph A. Konstan.

[65] Will Hill, Larry Stead, Mark Rosenstein, and George Furnas. Recommending and evaluating choices in a virtual community of use. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 194–201, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[66] Lorin M. Hitt and Prasanna (Sonny) Tambe. Broadband adoption and content consumption. *Information Economics and Policy*, 2007.

[67] Thomas Hofmann. Latent semantic models for collaborative filtering. *ACM Transactions on Information Systems*, 22(1):89–115, 2004.

[68] Thomas Hofmann and Jan Puzicha. Latent class models for collaborative filtering. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 688–693, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[69] John B. Horrigan. Home broadband adoption 2008. Technical report, Pew Internet & American Life Project, 2008.

[70] Zan Huang, Hsinchun Chen, and Daniel Zeng. Applying associative retrieval techniques to alleviate the sparsity problem in collaborative filtering. *ACM Transactions on Information Systems*, 22(1):116–142, 2004.

[71] Il Im and Alexander Hars. Does a one-size recommendation system fit all? the effectiveness of collaborative filtering based recommendation systems across different domains and search modes. *ACM Transactions on Information Systems*, 26(1):4, 2007.

[72] Anil K. Jain, Jianchang Mao, and K. Mohiuddin. Artificial neural networks: A tutorial. *IEEE Computer*, 29:31–44, 1996.

[73] Rong Jin and Luo Si. A study of methods for normalizing user ratings in collaborative filtering. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 568–569, New York, NY, USA, 2004. ACM.

[74] Rong Jin, Luo Si, and Chengxiang Zhai. Preference-based graphic models for collaborative filtering. In *Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence*, pages 329–336, 2003.

[75] George Karypis. Evaluation of item-based Top-N recommendation algorithms. In *cikm*, pages 247–254, 2001.

[76] Noriaki Kawamae and Katsumi Takahashi. Information retrieval based on collaborative filtering with latent interest semantic map. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 618–623, New York, NY, USA, 2005. ACM.

[77] Arnd Kohrs and Bernard Mérialdo. Improving collaborative filtering for new-users by smart object selection. In *ICME 2001, International Conference on Media Futures*, May 2001.

[78] A. N. Kolmogorov. Unbiased estimates. *Izv. Akad. Nauk SSSR Ser. Mat.*, 14(4):303–326, 1950.

[79] Joseph A. Konstan, Bradley N. Miller, David Maltz, Jonathan L. Herlocker, Lee R. Gordon, and John Riedl. GroupLens: Applying collaborative filtering to usenet news. *Communications of the ACM*, 40(3):77–87, 1997.

[80] Jonathan Koren, Yi Zhang, and Xue Liu. Personalized interactive faceted search. In *Proceedings of the 17th International Conference on World Wide Web*, pages 477–486, New York, NY, USA, 2008. ACM.

[81] Yehuda Koren. The BellKor solution to the netflix grand prize, August 2009.

[82] Bruce Krulwich and Chad Burkey. The infofinder agent: Learning user interests through heuristic phrase extraction. *IEEE Expert: Intelligent Systems and Their Applications*, 12(5):22–27, 1997.

[83] Chuck P. Lam. Collaborative filtering using associative neural memory. In Bamshad Mobasher and Sarabjot S. Anand, editors, *Itwp*, volume 3169 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2003.

[84] Ken Lang. Newsweeder: Learning to filter netnews. In *Proceedings of the 12th International Machine Learning Conference*, 1995.

[85] Neal Lathia, Stephen Hailes, and Licia Capra. Private distributed collaborative filtering using estimated concordance measures. In *Proceedings of the 2007 ACM Conference on Recommender Systems*, pages 1–8, New York, NY, USA, 2007. ACM.

[86] Neal Leavitt. Recommendation technology: Will it boost e-commerce? *Computer*, 39(5):13–16, 2006.

[87] Daniel Lemire. Scale and translation invariant collaborative filtering systems. *Information Retrieval*, 8(1):129–150, January 2005.

[88] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. The dynamics of viral marketing. In *Proceedings of the 7th ACM Conference on Electronic Commerce*, pages 228–237, New York, NY, USA, 2006. ACM.

[89] Cane Wing-ki Leung, Stephen Chi-fai Chan, and Fu-lai Chung. Integrating collaborative filtering and sentiment analysis: A rating inference approach. In *Proceedings of the Ecai 2006 Workshop on Recommender Systems*, pages 62–66, Riva del Garda, I, 2006.

[90] Cane Wing-ki Leung, Stephen Chi-fai Chan, and Fu-lai Chung. Applying cross-level association rule mining to cold-start recommendations. In *Proceedings of the 2007 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Workshops*, pages 133–136, Washington, DC, USA, 2007. IEEE Computer Society.

[91] Henry Lieberman. Letizia: an agent that assists web browsing. In *IJCAI'95: Proceedings of the 14th international joint conference on Artificial intelligence*, pages 924–929, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[92] Weiyang Lin, Sergio A. Alvarez, and Carolina Ruiz. Collaborative recommendation via adaptive association rule mining. In *Data Mining and Knowledge Discovery*, 2000.

[93] Weiyang Lin, Sergio A. Alvarez, and Carolina Ruiz. Efficient adaptive-support association rule mining for recommender systems. *Data Min. Knowl. Discov.*, 6(1):83–105, 2002.

[94] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[95] Hao Ma, Irwin King, and Michael R. Lyu. Effective missing data prediction for collaborative filtering. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 39–46, New York, NY, USA, 2007. ACM.

[96] Thomas W Malone, Kenneth R Grant, Franklyn A Turbak, Stephen A Brobst, and Michael D Cohen. Intelligent information-sharing systems. *Communications of the ACM*, 30(5):390–402, 1987.

[97] David Maltz and Kate Ehrlich. Pointing the way: Active collaborative filtering. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 202–209, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[98] David A. Maltz and David A. Maltz. Distributing information for collaborative filtering on usenet net news. Technical report, MIT Department of EECS MS Thesis, 1994.

[99] P. Massa. *Trust-Aware Decentralized Recommender Systems: PhD Research Proposal*. PhD thesis, University of Trento, May 2003.

[100] Darrell D. Massie. Neural network fundamentals for scientists and engineers. In *Proceedings of the International Congress on Efficiency, Costs, Optimization, Simulation and Environmental Aspects of Energy Systems and Processes*, pages 123–128, July 2001.

[101] Bhaskar Mehta, Thomas Hofmann, and Wolfgang Nejdl. Robust collaborative filtering. In *Proceedings of the 2007 ACM Conference on Recommender Systems*, pages 49–56, New York, NY, USA, 2007. ACM.

[102] Stuart E. Middleton. *Capturing Knowledge of User Preferences with Recommender Systems*. PhD thesis, Faculty of Engineering and Applied Science Department of Electronics and Computer Science, 2003.

[103] Bradley N. Miller, John T. Riedl, and Joseph A. Konstan. Experiences with GroupLens: Making usenet useful again. In *Proceedings of the 1997 Usenix Winter Technical Conference*, pages 219–231, 1997.

[104] Bradley Norman Miller. *Toward a Personal Recommender System*. PhD thesis, University of Minnesota, 2003.

[105] N. Miller, Bradley, A. Konstan, Joseph, and John Riedl. PocketLens: Toward a personal recommender system. *ACM Trans. Inf. Syst.*, 22(3):437–476, 2004.

[106] Sung-Hwan Min and Ingoo Han. Optimizing collaborative filtering recommender systems. In Piotr S. Szczepaniak, Janusz Kacprzyk, and Adam Niewiadomski, editors, *Advances in Web Intelligence*, volume 3528, pages 313–319, 2005.

[107] Bamshad Mobasher, Honghua Dai, Tao Luo, Miki Nakagawa, Yuqing Sun, and Jim Wiltshire. Discovery of aggregate usage profiles for web personalization. In *Proceedings of the WebKDD Workshop*, 2000.

[108] Alexandros Moukas and Pattie Maes. Amalthaea: An evolving multi-agent information filtering and discovery system for the WWW. *Autonomous Agents and Multi-Agent Systems*, 1(1):59–88, 1998.

[109] Ulrich Müller. Notes on sufficient statistics and the rao-blackwell theorem, 2008.

[110] K Nadiminti, MD De Assunção, and R Buyya. Distributed systems and recent innovations: Challenges and benefits. *InfoNet Magazine*, 16(3):1–5, 2006.

[111] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, pages 111–125. IEEE Computer Society, 2008.

[112] Tavi Nathanson, Ephrat Bitton, and Ken Goldberg. Eigentaste 5.0: Constant-time adaptability in a recommender system using item clustering. In *Proceedings of the 2007 ACM Conference on Recommender Systems*, pages 149–152, New York, NY, USA, 2007. ACM.

[113] Daniel Siaw Weng Ngu and Xindong Wu. Sitehelper: a localized agent that helps incremental exploration of the world wide web. *Comput. Netw. ISDN Syst.*, 29(8-13):1249–1255, 1997.

[114] D. Nguyen and B. Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Proceedings of the International Joint Conference on Neural Networks*, volume 3, pages 21–26, 1990.

[115] Hien Nguyen and Peter Haddawy. The decision-theoretic video advisor. In *AAAI-98 Workshop on Recommender Systems*, pages 77–80, 1998.

[116] John O'Donovan and Barry Smyth. Trust in recommender systems. In *Proceedings of the 10th International Conference on Intelligent User Interfaces*, pages 167–174, New York, NY, USA, 2005. ACM.

[117] Louis Olshevsky. Two properties of sufficient statistics. *The Annals of Mathematical Statistics*, 11(1):pp. 104–106, 1940.

[118] Panagiotis and Yannis Manolopoulos. Collaborative filtering: Fallacies and insights in measuring similarity, 2006.

[119] Andrew R. Pariser and Willard L. Miranker. Dimensionality reduction via self-organizing feature maps for collaborative filtering. In *International Joint Conference on Neural Networks*, pages 1941–1946. IEEE, 2007.

[120] Seung-Taek Park and David M. Pennock. Applying collaborative filtering techniques to movie search for better ranking and browsing. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 550–559, New York, NY, USA, 2007. ACM.

[121] Arkadiusz Paterek. Improving regularized singular value decomposition for collaborative filtering. In *Proc. KDD Cup Workshop at SIGKDD'07, 13th ACM Int. Conf. on Knowledge Discovery and Data Mining*, pages 39–42, 2007.

[122] P. Paulson and A. Tzanavari. Combining collaborative and content-based filtering using conceptual graphs. In *Words: Learning, Fusion, and Reasoning within a Formal Linguistic Representation Framework*, volume 2873, pages 168–185, 2003.

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

[124] David M. Pennock, Eric Horvitz, Steve Lawrence, and C. Lee Giles. Collaborative filtering by personality diagnosis: A hybrid memory and model-based approach. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 473–480, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[125] Gregory Piatetsky. Interview with simon funk. *ACM SIGKDD Explorations Newsletter*, 9(1):38–40, 2007.

[126] Gregory Piatetsky-Shapiro, Chabane Djeraba, Lise Getoor, Robert Grossman, Ronen Feldman, and Mohammed Zaki. What are the grand challenges for data mining?: KDD-2006 panel report. *ACM SIGKDD Explorations Newsletter*, 8(2):70–77, 2006.

[127] Martin Piotte and Martin Chabbert. The pragmatic theory solution to the netflix grand prize. Pragmatic Theory Inc., Canada, August 2009.

[128] Huseyin Polat and Wenliang Du. SVD-based collaborative filtering with privacy. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 791–795, New York, NY, USA, 2005. ACM.

[129] Rin Popescul, Lyle H. Ungar, David M. Pennock, and Steve Lawrence. Probabilistic models for unified collaborative and content-based recommendation in sparse-data environments. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 437–444, 2001.

[130] Seppo Puuronen and Alexey Tsymbal. Local feature selection with dynamic integration of classifiers. *Fundam. Inf.*, 47(1-2):91–117, 2001.

[131] Naren Ramakrishnan, Benjamin J. Keller, Batul J. Mirza, Ananth Y. Grama, and George Karypis. Privacy risks in recommender systems. *IEEE Internet Computing*, 5(6):54–62, 2001.

[132] Ralf Rantzau and Holger Schwarz. A multi-tier architecture for high-performance data mining. in: Buchmann, A. P. (ed.): Datenbanksysteme in Büro, technik und wissenschaft. Technical report, University of Stuttgart, 1999.

[133] Al M. Rashid, Shyong, George Karypis, and John Riedl. ClustKNN: A highly scalable hybrid model- & memory-based cf algorithm. In *WebKDD 2006*, Philadelphia, Pennsylvania, USA, August 2006.

[134] Al Mamunur Rashid, Istvan Albert, Dan Cosley, Shyong K. Lam, Sean M. McNee, Joseph A. Konstan, and John Riedl. Getting to know you: Learning new user preferences in recommender systems. In *Proceedings of the 7th International Conference on Intelligent User Interfaces*, pages 127–134, New York, NY, USA, 2002. ACM.

[135] Al Mamunur Rashid, George Karypis, and John Riedl. Influence in ratings-based recommender systems: An algorithm-independent approach. In *Sdm*, 2005.

[136] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. GroupLens: An open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, pages 175–186, New York, NY, USA, 1994. ACM.

[137] Paul Resnick and Rahul Sami. The influence limiter: Provably manipulation-resistant recommender systems. In *Proceedings of the 2007 ACM Conference on Recommender Systems*, pages 25–32, New York, NY, USA, 2007. ACM.

[138] Francesco Ricci and Hannes Werthner. Introduction to the special issue: Recommender systems. *Int. J. Electron. Commerce*, 11(2):5–9, 06-7.

[139] James Rucker and Marcos J. Polanco. Siteseer: Personalized navigation for the web. *Communications of the ACM*, 40(3):73–76, 1997.

[140] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th International Conference on Machine Learning*, pages 791–798, New York, NY, USA, 2007. ACM.

[141] J. J. Sandvig, Bamshad Mobasher, and Robin Burke. Robustness of collaborative recommendation based on association rule mining. In *Proceedings of the 2007 ACM conference on Recommender systems*, RecSys '07, pages 105–112, New York, NY, USA, 2007. ACM.

[142] Badrul Sarwar, George Karypis, Joseph Konstan, and John Reidl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, pages 285–295, New York, NY, USA, 2001. ACM.

[143] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Analysis of recommendation algorithms for e-commerce. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pages 158–167, New York, NY, USA, 2000. ACM.

[144] Badrul M. Sarwar, George Karypis, Joseph Konstan, and John Riedl. Recommender systems for large-scale e-commerce: Scalable neighborhood formation using clustering. Technical report, University of Minnesota, 2002.

[145] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John T. Riedl. Application of dimensionality reduction in recommender systems–a case study. In *ACM WebKDD Workshop*, 2000.

[146] Badrul M. Sarwar, Joseph A. Konstan, Al Borchers, Jon Herlocker, Brad Miller, and John Riedl. Using filtering agents to improve prediction quality in the GroupLens research collaborative filtering system. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, pages 345–354, New York, NY, USA, 1998. ACM.

[147] J. Ben Schafer, Joseph Konstan, and John Riedi. Recommender systems in E-Commerce. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, pages 158–166, New York, NY, USA, 1999. ACM.

[148] Upendra Shardanand and Patti Maes. Social information filtering: Algorithms for automating "word of mouth". In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 1, pages 210–217, 1995.

[149] B. Sheth and P. Maes. Evolving agents for personalized information filtering. In *Proceedings Of the Ninth Conference on Artificial Intelligence for Applications*, pages 345–352, Orlando, FL, 1993.

[150] Ian M. Soboroff and Charles K. Nicholas. Combining content and collaboration in text filtering. In *Proceedings of the IJCAI'99 Workshop on Machine Learning for Information Filtering*, pages 86–91, 1999.

[151] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[152] Xiaoyuan Su, Russell Greiner, Taghi M. Khoshgoftaar, and Xingquan Zhu. Hybrid collaborative filtering algorithms using a mixture of experts. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pages 645–649, Washington, DC, USA, 2007. IEEE Computer Society.

[153] Xiaoyuan Su, Taghi M. Khoshgoftaar, Xingquan Zhu, and Russell Greiner. Imputation-boosted collaborative filtering using machine learning classifiers. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 949–950, New York, NY, USA, 2008. ACM.

[154] S. Shyam Sundar, Anne Oeldorf-Hirsch, and Qian Xu. The bandwagon effect of collaborative filtering technology. In *Extended Abstracts on Human Factors in Computing Systems*, pages 3453–3458, New York, NY, USA, 2008. ACM.

[155] Shyam S. Sundar. The MAIN Model: A Heuristic Approach to Understanding Technology Effects on Credibility. *The John D. and Catherine T. MacArthur Foundation Series on Digital Media and Learning*, -:73–100, December 2007.

[156] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. Addison-Wesley, 2007.

[157] Loren Terveen, Will Hill, Brian Amento, David McDonald, and Josh Creter. PHOAKS: A system for sharing recommendations. *Communications of the ACM*, 40(3):59–62, 1997.

[158] N. A. Thacker. Tutorial: Supervised neural networks in machine vision, December 1998.

[159] F. Tichy, W. A catalogue of general-purpose software design patterns. In *TOOLS '97: Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems*, page 330, Washington, DC, USA, 1997. IEEE Computer Society.

[160] Andreas Tošcher, Michael Jahrer, and Robert M. Bell. The BigChaos solution to the netflix grand prize, September 2009.

[161] Karen H. L. Tso-Sutter, Leandro Balby Marinho, and Lars Schmidt-Thieme. Tag-aware recommender systems by fusion of collaborative filtering algorithms. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 1995–1999, New York, NY, USA, 2008. ACM.

[162] L. Ungar and D. Foster. Clustering methods for collaborative filtering. In *Proceedings of the Workshop on Recommendation Systems*. AAAI Press, Menlo Park California, 1998.

[163] Jun Wang, Arjen P. de Vries, and Marcel J. T. Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 501–508, New York, NY, USA, 2006. ACM.

[164] Jun Wang, Arjen P. de Vries, and Marcel J. T. Reinders. Unified relevance models for rating prediction in collaborative filtering. *ACM Transactions on Information Systems*, 26(3):1–42, 2008.

[165] Jun Wang, Johan Pouwelse, Reginald L. Lagendijk, and Marcel J. T. Reinders. Distributed collaborative filtering for peer-to-peer file sharing systems. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1026–1030, New York, NY, USA, 2006. ACM.

[166] Jun Wang, Marcel J. T. Reinders, Reginald L. Lagendijk, and Johan Pouwelse. Self-organizing distributed collaborative filtering. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 659–660, New York, NY, USA, 2005. ACM.

[167] Ahmad M. Ahmad Wasfi. Collecting user access patterns for building user profiles and collaborative filtering. In *Proceedings of the 4th International Conference on Intelligent User Interfaces*, pages 57–64, New York, NY, USA, 1999. ACM.

[168] Jianshu Weng, Chunyan Miao, and Angela Goh. Improving collaborative filtering with trust-based metrics. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1860–1864, New York, NY, USA, 2006. ACM.

[169] Bo Xie, Peng Han, Fan Yang, Rui-Min Shen, Hua-Jun Zeng, and Zheng Chen. DCFLA: A distributed collaborative-filtering neighbor-locating algorithm. *Information Sciences*, 177(6):1349–1363, March 2007.

[170] Jin-Min Yang and Kin Fun Li. An adaptive user-genre-item model for collaborative filtering. In *IEEE Pacific Rim Conference on Communications Computers and Signal Processing*, pages 257–262, 2007.

[171] Kai Yu, Anton Schwaighofer, Volker Tresp, Xiaowei Xu, and Hans-Peter Kriegel. Probabilistic memory-based collaborative filtering. *IEEE Transactions on Knowledge and Data Engineering*, 16(1):56–69, 2004.

[172] Sheng Zhang, James Ford, and Fillia Makedon. A privacy-preserving collaborative filtering scheme with two-way communication. In *Proceedings of the 7th ACM Conference on Electronic Commerce*, pages 316–323, New York, NY, USA, 2006. ACM.

[173] Yi Zhang, Jamie Callan, and Thomas Minka. Novelty and redundancy detection in adaptive filtering. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 81–88, New York, NY, USA, 2002. ACM.

[174] Yi Zhang and Jonathan Koren. Efficient bayesian hierarchical user modeling for recommendation system. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 47–54, New York, NY, USA, 2007. ACM.

[175] Cai-Nicolas Ziegler. Applying feed-forward neural networks to collaborative filtering. Master's thesis, Universität Freiburg, 2006.