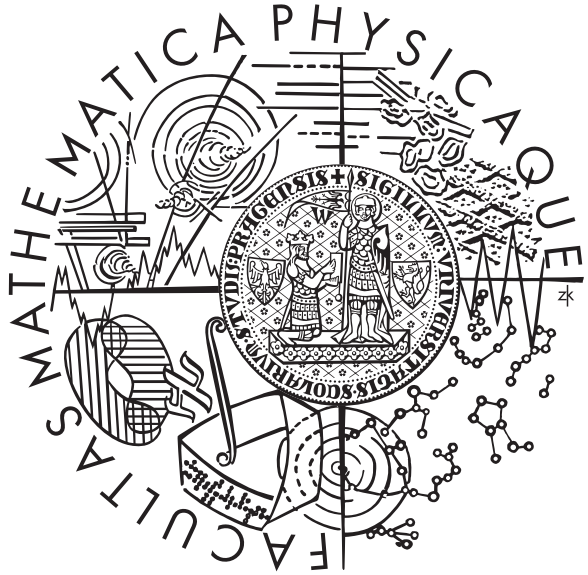
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Lukáš Brožovský

Recommender System for a Dating Service

Department of Software Engineering

Supervisor: RNDr. Václav Petříček

Study program: Computer Science

First and foremost I would like to thank my advisor Václav Petříček for his great support and priceless comments, especially on the text of the thesis.

For their generous assistance in providing the original test data sets, I would like to acknowledge the administrator of the LibimSeTi.cz web application, Mr. Oldřich Neuberger, and also the administrator of the ChceteMe.volny.cz web application, Mr. Milan Salajka.

My thanks as well to my family and to all who have patiently listened and supported me during my work on the thesis, namely to: Ján Antolík, František Brantál, Eva Daňhelková, Luboš Lipinský and Miroslav Nagy.

I declare that I wrote the master thesis by myself, using only referenced sources.
I agree with lending the thesis.

Prague, August 9, 2006 *Lukáš Brožovský*

Contents

Contents i

List of Figures iii

List of Tables iv

1 Introduction 1
- 1.1 Collaborative Filtering, Recommender Systems . . . 1
  - 1.1.1 Existing Applications . . . 2
- 1.2 Dating Services . . . 3
- 1.3 Contribution . . . 5
- 1.4 Outline . . . 5
- 1.5 Notation . . . 5

2 Data Sets Analysis 7
- 2.1 MovieLens Data Set . . . 7
- 2.2 Jester Data Set . . . 8
- 2.3 ChceteMě Data Set . . . 8
- 2.4 LíbímSeTi Data Set . . . 9
- 2.5 Data Sets Comparison . . . 9
  - 2.5.1 Rating Frequencies . . . 10
  - 2.5.2 Value Frequencies . . . 11
  - 2.5.3 Similarity Frequencies . . . 13

3 Algorithms 17
- 3.1 Random Algorithm . . . 17
- 3.2 Mean Algorithm . . . 18
- 3.3 User-User Nearest Neighbours Algorithm . . . 18
  - 3.3.1 Pearson's Correlation Coefficient . . . 19
- 3.4 Item-Item Nearest Neighbours Algorithm . . . 19
  - 3.4.1 Adjusted Pearson's Correlation Coefficient . . . 20

4 ColFi Architecture **21**
4.1 ColFi Architecture Overview . . . 21
4.2 Client-Server Solution . . . 22
4.3 Data Manager . . . 22
4.4 ColFi Services . . . 22
4.5 Algorithm Service . . . 23

5 ColFi Implementation **24**
5.1 ColFi Server . . . 25
5.1.1 Logging . . . 25
5.1.2 Configuration . . . 25
5.1.3 Data Manager Instance . . . 27
5.1.4 ColFi Service Instances . . . 27
5.1.5 ColFi Communication Protocol . . . 28
5.2 Data Managers . . . 28
5.2.1 Cached Matrix and Cached Data Manager Adapter . . . 30
5.2.2 Empty Data Manager . . . 32
5.2.3 File Data Manager . . . 32
5.2.4 Cached MySQL Data Manager . . . 33
5.3 ColFi Services . . . 34
5.3.1 Algorithm Service . . . 35
5.4 Algorithms . . . 36
5.4.1 Random Algorithm . . . 36
5.4.2 Mean Algorithm . . . 36
5.4.3 User-User Nearest Neighbours Algorithm . . . 37
5.4.4 Item-Item Nearest Neighbours Algorithm . . . 39

6 Benchmarks **41**
6.1 Simulation Benchmarks . . . 41
6.1.1 Simulation Benchmarks Design . . . 42
6.1.2 Simulation Benchmarks Results . . . 45
6.2 Empirical Benchmarks . . . 58
6.2.1 Empirical Benchmarks Design . . . 58
6.2.2 Empirical Benchmarks Results . . . 58

7 Conclusion **61**
7.1 Contribution . . . 61
7.2 Future Work . . . 62

A Detailed GivenRandomX Analysis **63**

Bibliography **64**

List of Figures

2.1 ChceteMě sessions 9
2.2 Ratings frequency diagrams 11
2.3 Values frequency diagrams 12
2.4 Distributions of rating similarities 14
2.5 Mean sizes of rating overlap for different similarities 14
2.6 Distributions of score similarities 15
2.7 Mean sizes of score overlap for different similarities 15
2.8 Distributions of score adjusted similarities 16
2.9 Mean sizes of rating overlap for different adjusted similarities 16

4.1 ColFi architecture 21

5.1 `DataManager` interface 29
5.2 `DataManagerListener` interface 30
5.3 Cached matrix implementation 31
5.4 `DataManager` implementations hierarchy 33
5.5 `Service` interface 34
5.6 `Algorithm` interface 35
5.7 `Algorithm` implementations hierarchy 40

6.1 AllButOne strategy pseudo-code 43
6.2 GivenRandomX strategy pseudo-code 44
6.3 GivenRandomX strategy benchmark results - MovieLens 49
6.4 GivenRandomX strategy benchmark results - Jester 50
6.5 GivenRandomX strategy benchmark results - ChceteMě 51
6.6 GivenRandomX strategy benchmark results - LíbímSeTi 52
6.7 Production strategy benchmark results - MovieLens 54
6.8 Production strategy benchmark results - Jester 55
6.9 Production strategy benchmark results - LíbímSeTi 56
6.10 LíbímSeTi experiment 59

List of Tables

1.1 Example online dating services based on rating of user profiles 4

2.1 Data sets overview 10

6.1 AllButOne strategy benchmark results 47
6.2 Production strategy benchmark results 57
6.3 Empirical benchmark results 60

Název práce: Doporučovací systém v online seznamce

Autor: Lukáš Brožovský

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Václav Petříček

E-mail vedoucího: v.petricek@gmail.com

Abstrakt: Hlavním cílem diplomové práce je ověřit využitelnost doporučovacích systémů založených na kolaborativním filtrování pro oblast online seznamek. Součástí práce je vlastní implementace několika nejběžnějších algoritmů kolaborativního filtrování a systému, který bude doporučovat uživatelům potenciální kandidáty pro seznámení na základě jejich preferencí, hodnocení fotografií ostatních uživatelů apod. Princip kolaborativního filtrování je založen na předpokladu, že pokud byla hodnocení dvou uživatelů doposud stejná či podobná, bude tomu tak i v budoucnu.

Druhá část práce obsahuje několik testů a porovnání výkonnosti a přesnosti implementovaného systému na veřejně dostupných datech (MovieLens a Jester) a také na datech pocházejících přímo z oblasti online seznamek (ChceteMě a LíbímSeTi). Výsledky testů prokazují možnost úspěšné aplikace kolaborativního filtrování v oblasti online seznamek.

Klíčová slova: doporučovací systém, kolaborativní filtrování, seznamování, online seznamka, algoritmus (k)-nejbližších sousedů, Pearsonův korelační koeficient, Java

Title: Recommender System for a Dating Service

Author: Lukáš Brožovský

Department: Department of Software Engineering

Supervisor: RNDr. Václav Petříček

Supervisor's e-mail address: v.petricek@gmail.com

Abstract: The aim of the thesis is to research the utility of collaborative filtering based recommender systems in the area of dating services. The practical part of the thesis describes the actual implementation of several standard collaborative filtering algorithms and a system, which recommends potential personal matches to users based on their preferences (e.g. ratings of other user profiles). The collaborative filtering is built upon the assumption, that users with similar rating patterns will also rate alike in the future.

Second part of the work focuses on several benchmarks of the implemented system's accuracy and performance on publicly available data sets (MovieLens and Jester) and also on data sets originating from real online dating services (ChceteMě and LíbímSeTi). All benchmark results proved that collaborative filtering technique could be successfully used in the area of online dating services.

Keywords: recommender system, collaborative filtering, matchmaking, online dating service, (k)-nearest neighbours algorithm, Pearson's correlation coefficient, Java

Revision: 1.0 (August 9, 2006 7:49pm)

Chapter 1

Introduction

This chapter makes the reader acquainted with the concept of collaborative filtering and introduces its utilization in the environment of dating services.

Since long before the industrial revolution, *information* has been the most valuable element in almost all human activities. For the last five decades, with the great expansion of technology, and especially with the Internet boom over the last few years, the amount of readily available information has increased exponentially. Objectively, searching through the vast amount of available information has become a true challenge for everyone. This phenomenon, mostly referred to as the *information overload*, describes the state of having too much information to make relevant decisions or remain informed about a certain topic [2].

The task of making well-informed decisions supported by finding relevant information is addressed in research of the domain of *information retrieval*, which is "... the art and science of searching for information in documents, searching for documents themselves, searching for metadata which describe documents, or searching within databases, whether relational stand alone databases or hypertext networked databases such as the Internet or intranets, for text, sound, images or data" [3].

The research on information retrieval has developed several technology-based solutions addressing information overload: intelligent agents, ranking algorithms, cluster analysis, web mining/data mining, personalization, recommender systems and collaborative filtering [10]. This thesis focuses on recommender systems relying on collaborative filtering.

1.1 Collaborative Filtering, Recommender Systems

Typically, a search for a specific piece of information is based on some kind of a query supplied by a user. For example, a search for textual documents or web pages is mostly performed with a simple query containing words or phrases desired in the returned documents. In search for audio files a structured query might be used, combining for example song title, artist, album name, genre and release date. These examples fall within the

category of the *content-based searches* (also called the *retrieval approach*).

In contrary to the content-based searches, there exists a different technique called *information filtering* (employed by information filtering systems). Generally, the goal of an information filtering system is to sort through large volumes of dynamically generated information and present to the user those which are likely to satisfy his or her information requirement [27]. Information filtering does not have to be based on an explicit user query. A good example of information filtering from every day life could be the filtering of news web sites, research articles or e-mails.

Closely related to information filtering is the concept of **recommender system**. Recommender system is a system or a program which attempt to predict information/items (movies, music, books, news, web pages or people) that a user may be interested in. These systems often use both, query supported retrieval approaches and information filtering approaches based on information about the user's profile. This need of utilization of user profiles frequently classifies the recommender systems as the *personalized information systems*.

Collaborative filtering represents the information filtering approach, used by many present-day recommender systems. The main idea of collaborative filtering is to select and recommend items of interest for a particular user based on opinions of other users in the system. This technique relies on a simple assumption, that a good way to find interesting content is to identify other people with similar interests, and then select or recommend items that those similar people like.

Someone browsing an e-commerce web site might not always have a specific request and so it is the web site's interest to provide compelling recommendations to attract customers. For example, a web interface of e-shop with audio CDs could store the set of CDs particular user viewed during the session and then recommend to him/her albums bought by other users browsing around similar set of CDs (e.g., similar titles, genres) [24].

1.1.1 Existing Applications

One of the earliest implementations of collaborative filtering approach was used in the Tapestry experimental mail system developed at the Xerox Palo Alto Research Center in the year 1992 [17]. This system was used as a mail filtering information system utilizing the collaborative sense of a small group of participating users. The Tapestry system did not use a definite concept of *automated collaborative filtering*[1] as the actual filtering technique was strictly bound to the concrete user-defined queries. Every Tapestry user maintained personal list of filtering queries based on SQL-like language – Tapestry Query Language (TQL). The collaborative part was included as an extension to TQL, allowing users to link their queries to the queries of other specific users. For example, a query returning all messages selected by Terry's "Baseball" query that contain the word "Dodgers" would be: *m IN Terry.Baseball AND m.words = ('Dodgers')*.

[1] Automated collaborative filtering – collaborative filtering where the system automatically selects the co-users whose opinions are used

The concept of real automated collaborative filtering first appeared in the GroupLens Research Project[2] system using neighborhood-based algorithm for providing personalized predictions for Usenet news articles [31]. Since then, the research on collaborative filtering has drawn significant attention and a number of collaborative filtering systems in broad variety of application domains have appeared.

Probably the most publicly aware system using collaborative filtering in the last few years has been the item-to-item recommender at Amazon.com. It is used to personalize the online store for each customer. The store radically changes based on customer interests, showing programming titles to a software engineer and baby toys to a new mother [24].

More examples of commercial systems using collaborative filtering are:

- NetFlix.com – the world's largest online DVD movie rental service
- GenieLab::Music – a music recommender system (compatible with iTunes)
- Musicmatch Jukebox – a music player, organizer and recommendation system
- AlexLit.com – Alexandria Digital Literature, eBookstore under the Seattle Book
- Findory.com – personalized news website

And the examples of non-commercial systems using collaborative filtering are:

- inDiscover.net – music recommendation site (formerly RACOFI Music)
- GiveALink.org – a public site where people can donate their bookmarks
- Gnod.net – the Global Network of Dreams, a recommender system for music, movies and authors of books
- FilmAffinity.com – a movie recommender system

1.2 Dating Services

Wikipedia definition of *dating service* (or *dating system*) states: "A dating system is any systemic means of improving matchmaking via rules or technology. It is a specialized meeting system where the objective of the meeting, be it live or phone or chat based, is to go on a live date with someone, with usually romantic implications" [1]. The *matchmaking* is any expert-run process of introducing people for the purposes of dating and mating, usually in the context of marriage [4].

Typically, with most dating or matchmaking services, user has to waste considerable time filling out lengthy personality questionnaires and character trait analyzation charts as the matching phase of the service is strictly content-based. Most of the time the user

[2]GroupLens Research Project – http://www.cs.umn.edu/Research/GroupLens/

has to simply browse through thousands of user profiles returned by some complex query-based search (a user might be for example interested in blue-eyed blondes with university education). And this situation is the same for both online and offline dating services and also for both free and commercial dating services.

Most of the commercial sites let users register for free and bill them later for information on contacts to other users. Some examples of dating services are:

- America's Internet Dating – http://www.americasinternetdating.com/
- Date.com – http://www.date.com/
- Match.com – http://www.match.com/
- Perfect Match – http://www.perfectmatch.com/
- Grand – http://www.videoseznamka.cz/
- LíbímSeTi – http://www.libimseti.cz/

In the last few years, a specific subgroup of online dating service web applications has emerged on the Internet. Their main idea is to let users store personal profiles and then browse and **rate** other users' profiles[3]. Usually, the registered user is presented with random profiles (or with random profiles preselected with a simple condition - for example only men of certain age) and rates the profiles on a given numerical scale.

Table 1.1 shows the example online dating services based on the rating of user profiles:

Server	M users	F users	M pictures	F pictures	Ratings
chceteme.volny.cz	2,277				48,666
libimseti.cz	194,439				11,767,448
podivejse.cz	3,459		5,858		805,576
ukazse.cz	-				
vybersi.net	N/A		916	2,321	43,357
qhodnoceni.cz	8,440	2,520	12,482	6,843	N/A
palec.cz	-				
facka.com	-				
tentato.sk	N/A		663	353	-

Table 1.1: Example online dating services based on rating of profiles in the Czech Republic

Overview of Czech online dating services based on the rating of user profiles and their numbers of male/female users, male/female photographs and votes. In this study, the data sets for Chceteme.volny.cz (ChceteMě) and Libimseti.cz (LíbímSeTi) were available.

[3]Typically, user profiles are based on user photo(s) or sometimes on short audio/video records.

None of these applications uses any kind of sophisticated matchmaking or recommender system. They are all query oriented allowing users to use only simple filters and then browse the selected user profiles. To date and to my knowledge, there is only one and relatively new online dating service web application using profile rating that claims to use collaborative filtering for matchmaking - ReciproDate.com (http://www.reciprodate.com/). Their software is proprietary.

1.3 Contribution

The attempts on application of automated collaborative filtering to the online dating services have been very limited even though dating service is one of the often cited examples of its possible applications [7, 36, 5]. This thesis therefore focuses on the application of collaborative filtering technique in the domain of the online dating services based on the rating of user profiles. It appears that most users of dating services have a common need of recommendations based on a specific elusive criteria. Not even a complex query would capture the preferences of a regular user. Moreover, with the growing number of users registered in these online applications, it is no longer possible for them to browse through the huge amount of unsorted search results returned by any kind of query.

It is apparent, that the massive data sets of ratings that were gathered from the users of the dating service systems could be used as the input for any collaborative filtering algorithm to either directly recommend matching users or at least sort the results of any content-based search.

In this thesis, an open source recommender system (the **ColFi System**) is implemented, employing several basic collaborative filtering algorithms. The second part of the thesis presents benchmarks of these algorithms on the real online dating service data sets.

1.4 Outline

The rest of the thesis is organized in six remaining chapters. In Chapter 2, four experimental data sets are described and analysed in detail. Then, chapter 3 introduces basic collaborative filtering algorithms. Chapter 4 and chapter 5 present the ColFi System architecture and implementation respectively (these chapters are developer oriented). The benchmarking methodology and results are discussed in chapter 6. The concluding chapter 7 summarizes the contributions of the thesis and proposes possible directions for future work.

1.5 Notation

Throughout the following chapters of the thesis, a common basic concepts and typographic notation is used.

The typical data model consists of N users $1 \dots N$ and the complete ratings matrix $\mathbb{R}$. The ratings matrix is a two-dimensional matrix, where $r_{i,j} \in \mathbb{R}$ denotes the rating given by a user i to a user j (also called the score for a user j obtained from a user i). A single row of the ratings matrix is referred to as the ratings of particular user and represents all the ratings given by that user. A single column of the ratings matrix is referred to as the scores for particular user and represents all the scores obtained by that user. The total number of all non-empty (non-null) individual ratings ($\sim$ votes) in the ratings matrix is $R = |\mathbb{R}|$. All the ratings matrix values are represented in a discrete ratings scale $\langle r_{min}..r_{max} \rangle$.

Every Java™ related text (e.g., java package names, classes, methods and code listings) are typeset in `teletype` font.

Chapter 2

Data Sets Analysis

This chapter describes in detail several data sets used to develop and evaluate ColFi collaborative filtering algorithms.

Collaborative filtering is not so interesting without any reasonably large data set. Four independent experimental data sets were chosen. These data sets were used to evaluate the emergent collaborative filtering algorithms during the design and development phase. At the end, these data sets were also used to benchmark the final implementations - see chapter 6 on the page 41.

The data sets are based on the explicit data of various applications. Most of these applications do store other information along with the plain ratings matrix, but these additional data were discarded as they are out of the scope of this thesis. Some of the original data had to be modified in certain way to satisfy the requirements of the dating service application data set. These modifications were specific for each data source and they are described individually in the following paragraphs.

2.1 MovieLens Data Set

The MovieLens data set is the most popular publicly available experimental data set. The MovieLens is a web-based recommender system for movies[1], based on GroupLens technology (part of the GroupLens Research Project). The MovieLens data set originated from the EachMovie movie recommender data provided by HP/Compaq Research (formerly DEC Research).

In this thesis, the so called *1 Million MovieLens data set* was used in its full extent. Both MovieLens users and MovieLens movies were declared as ColFi users to conform the square ColFi ratings matrix. The original ratings scale $\langle 1..5 \rangle$ was linearly scaled into the ColFi's internal scale $\langle 1..127 \rangle$. This way transformed data set is further referred to as the **MovieLens data set**.

[1] MovieLens web page – http://movielens.umn.edu/

2.2 Jester Data Set

The Jester data set is another well known and publicly available experimental data set. The Jester Joke Recommender System is a web-based recommender system for jokes[2], created by Prof. Ken Goldberg and the Jester team at the Alpha Lab, UC Berkeley. The Jester data set contains continuous ratings of jokes collected between April 1999 – May 2003.

In this thesis, the complete Jester data set was used. Both Jester users and Jester jokes were declared as ColFi users to conform the square ColFi ratings matrix. The original continuous ratings scale $\langle -10..10 \rangle$ was linearly adjusted into the ColFi's internal discrete scale $\langle 1..127 \rangle$. This way transformed data set is further referred to as the **Jester data set**.

This data set is very specific due to the fact that it originally contained only 100 jokes which all the Jester users may have rated. This resulted in "almost empty" square ratings matrix with very dense 100 columns.

2.3 ChceteMě Data Set

The ChceteMě data set is based on the production data of the ChceteMě web application[3]. This web application lets users store their photographs in the system to be rated by other users. Photographs are not limited to the user's portraits (people store pictures of their pets, cars, etc.), so this web is not a straight forward dating service application. On the other hand, the main concept is close enough.

The original data did not include explicit identity of the rating users. The only thing application stored is the target user, rating value, time of the rating and the IP address of the rater. This information was used to scan for possible user *sessions* as follows: the session was declared as a set of ratings initiated from the same IP address and not having the idle period between any two contiguous ratings greater then T minutes. The figure 2.1 reflects the dependence of the number of unique sessions found based on the value of the parameter T. Ultimately the value $T = 15$ minutes was selected, according to the graph and the fact that the web application automatically logs any idle user out after exactly 15 minutes. This approach could never be absolutely accurate as one user might log in from different computers with different IP addresses or there might be multiple users rating simultaneously from behind the proxy server and so on.

Both original target users and found sessions were declared as ColFi users to conform the square ColFi ratings matrix. The original ratings scale $\langle 0..10 \rangle$ was linearly adjusted into the ColFi's internal scale $\langle 1..127 \rangle$. This way transformed data set is further referred as the **ChceteMě data set**.

Anonymized version of the final data set is publicly available as a part of the ColFi Project in the file named `colfi-dataset-chceteme.ser`.

[2] Jester Joke Recommender System web page – http://shadow.ieor.berkeley.edu/humor/

[3] ChceteMě web application – http://chceteme.volny.cz/

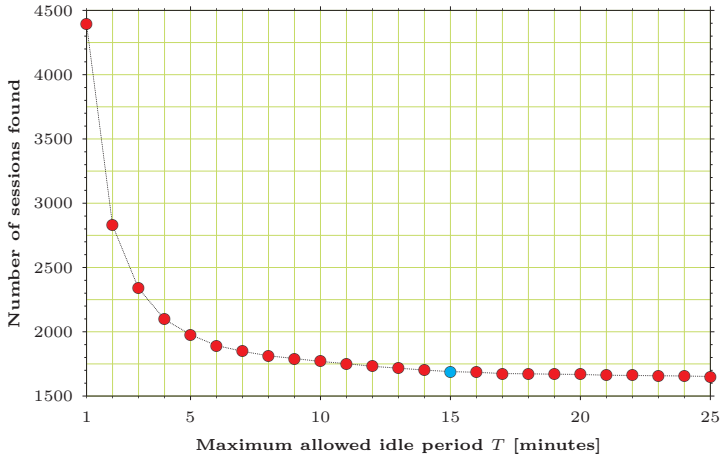


Figure 2.1: ChceteMě sessions

The number of unique sessions found in the original ChceteMě data based on the parameter T. The selected value $T = 15$ minutes is highlighted ($\sim$ 1,690 sessions $\sim$ 2,277 ColFi users).

2.4 LíbímSeTi Data Set

The LíbímSeTi data set is the most important data set through out this thesis as it is the target data set ColFi System was initially designed for. It consists of real online dating service data based on the LíbímSeTi web application[4].

In this thesis, a snapshot of the LíbímSeTi production database (ratings matrix data only) from July 2005 was used. The original ratings scale $\langle 1..10 \rangle$ was linearly scaled into the ColFi's internal scale $\langle 1..127 \rangle$. The original data set distinguishes between the *not rated* and the *not observed* - this extra information was discarded and both states were interpreted as *not observed*. This way transformed data set is further referred to as the **LíbímSeTi data set**.

Anonymized version of the final data set is publicly available as a part of the ColFi Project in the file named `colfi-dataset-libimseti.ser`.

2.5 Data Sets Comparison

Although all the mentioned data sets have a common structure after the appropriate conversions, there are certain major differences between them as the original data had each their own specificity. Table 2.1 summarizes the basic characteristics of all the data sets (their sizes, sparsity, values, etc.).

This table shows some interesting facts. First, not so surprisingly, the data set sizes are quite different and the ratings matrix density tends to be smaller for data sets containing more users (although this observation might be influenced by a very specific Jester data

[4]LíbímSeTi web application – http://www.libimseti.cz/

	MovieLens	**Jester**	**ChceteMě**	**LíbímSeTi**
Inception date	February 2003	May 2003	June 24, 2004	July 26, 2005
Total number of users	9,746	73,521	2,277	194,439
Number of active users	6,040	73,421	1,806	128,394
Number of passive users	3,706	100	499	120,393
Total number of ratings	**1,000,209**	**4,136,360**	**48,666**	**11,767,448**
Ratings matrix density	10.5302 ‰	0.7652 ‰	9.3864 ‰	0.3113 ‰
Max. ratings from 1 user	2,314	100	445	15,060
Max. scores for 1 user	3,428	73,413	354	69,365
Mean rating value	82.20	68.71	70.19	63.84
Median rating value	95	73	77	57
Standard deviation	35.05	33.62	42.73	46.26

Table 2.1: Data sets overview

An overview of the data set's basic characteristics. An *active user* is a user who have rated another user at least once. And *passive users* are users that have been scored at least once. The last three rows contain values characteristics in the ColFi's internal ratings scale ⟨1..127⟩. The *ratings matrix density* is expressed as a ratings matrix saturation (in ‰).

set). The second fact observed is the significant difference in the standard deviation of the ratings values between data sets which originated from the dating services data (ChceteMě and LíbímSeTi) and non-dating services data sets (MovieLens and Jester). This can be explained by rather dissimilar ratings value frequencies among these data sets as shown in the section 2.5.2 on the page 11.

2.5.1 Rating Frequencies

The ratings frequencies (or the ratings frequency diagrams), as shown in the figure 2.2, analyse the exact number of ratings given by each user. The diagrams show the total number of users per specific number of ratings given for all data sets. The absolute number of users depends on the particular data set size, so it is more interesting to compare the relative distributions among the data sets. It is obvious that there are more users with little number of ratings given. Those are the users who have rated few items/users and probably did not come back later to rate more. This observation is not so evident at first sight on the graphs for MovieLens and Jester data set, because there are missing values for low numbers of ratings given. That is caused by a data pre-selection performed on the original data by their authors[5]. Nevertheless, even for those data sets, the number of users grows for smaller numbers of ratings given and this tendency might be probably assumed

[5]Only such users who have rated at least 20 movies were selected into the MovieLens original data set. The Jester data set had a similar pre-selection condition - only users who have rated at least 15 jokes were selected.

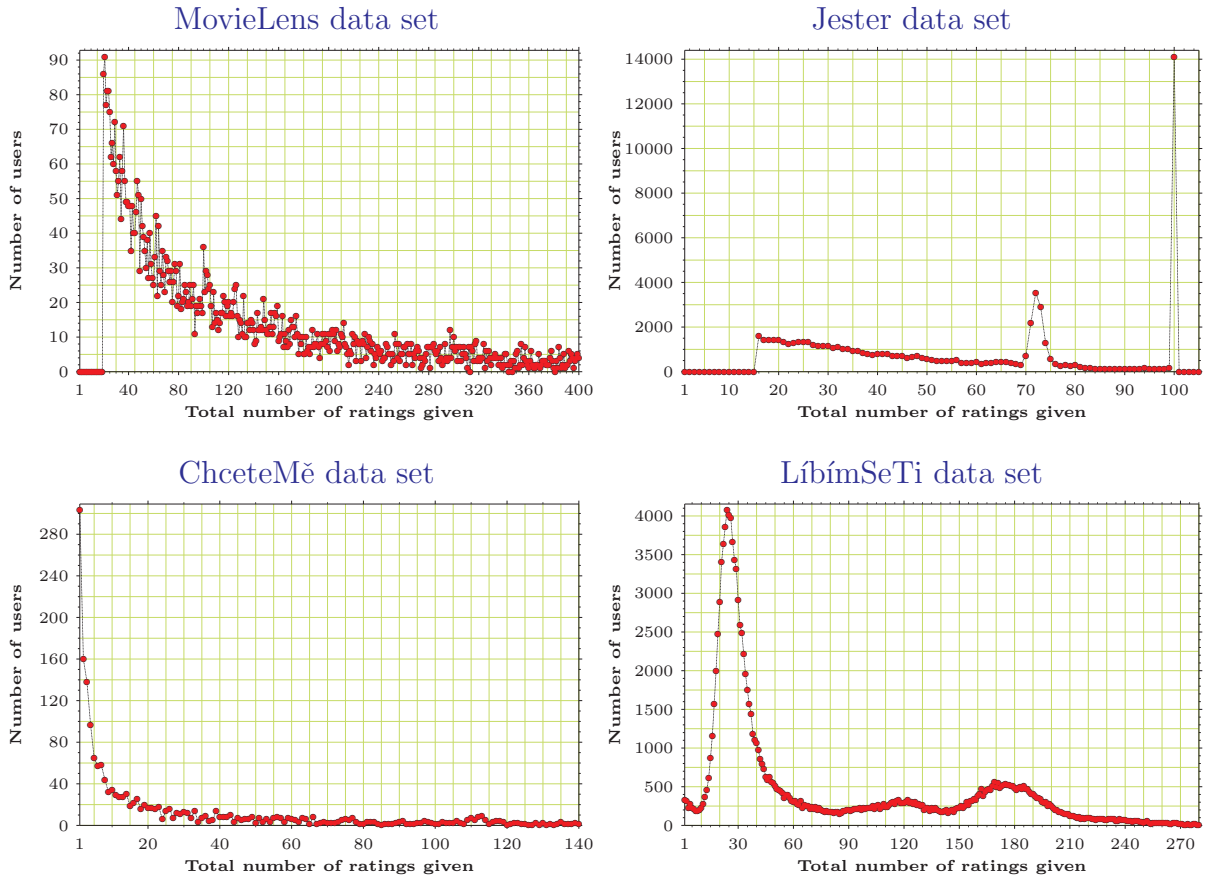


Figure 2.2: Ratings frequency diagrams

These frequency diagrams show the total number of users per specific number of ratings given for all data sets. The y-axis scale is dependent on the particular data set size.

also for the cropped data.

The Jester data set frequency diagram shows two more facts. First, there is majority of users who have rated all 100 jokes in the original data set. And second, there exists a strong group of users who have rated between 70 and 75 jokes. This is just a data set's specific feature and it has nothing to do with similar peak(s) in the frequency diagram of the LíbímSeTi data set.

2.5.2 Value Frequencies

The values frequencies (or the values frequency diagrams), as shown in the figure 2.3, analyse the different ratings values given by users.

The graph for the Jester data set is greatly influenced by the original "continuous" ratings scale. Although the ratings scale really was continuous, users of the Jester web application were limited to discrete values due to the actual implementation. This fact,

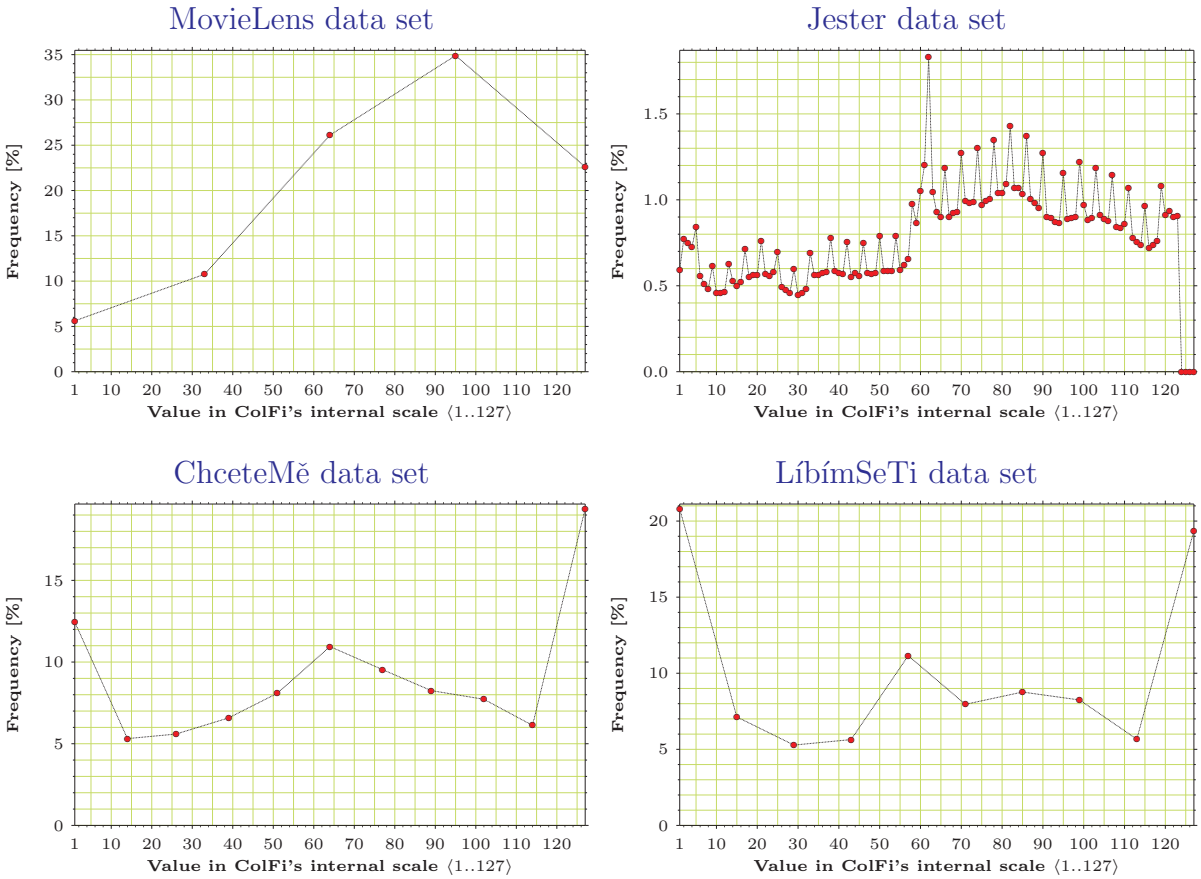


Figure 2.3: Values frequency diagrams

These frequency diagrams show the relative frequencies of the different ratings values in the internal ColFi's scale ⟨1..127⟩. The number of distinct values in each graph is influenced by the size of the original data ratings scale.

together with rounding issues during the data set rescaling process, led to the regular local peaks seen in the diagram.

Despite these irregularities, there is significant similarity between MovieLens and Jester data set and between ChceteMě and LíbímSeTi data set. Again this is probably caused by the fact that the first two data sets did not originate from the dating service data, in contrary to the ChceteMě and LíbímSeTi data sets.

It is interesting that users of dating services favour the minimum and the maximum value of the ratings scale. This is probably because of the fact that todays collaborative filtering dating services are mainly based on users photographs and because of that some people tend to rate in binary manner (like/dislike) and choose the scale's boundary values as their opinion.

2.5.3 Similarity Frequencies

The main concept of the collaborative filtering algorithms is to utilize the relative similarities[6] between users' ratings or scores. The similarity frequencies (or the similarities frequency diagrams) show the distributions of similarities for individual data sets.

Ratings Similarity Frequencies

The ratings similarity frequencies, as illustrated in the figure 2.4, display the distributions of similarities among users' ratings vectors (the rows of the ratings matrix). Six different distributions are presented for each data set to see the impact of the required minimum number of common votes in the vectors (which is a parameter of the similarity metric[7]).

The distribution for the Jester data set for *at least 5 common items* and for *at least 10 common items* turned out to be almost the same. That is because the number of ratings vector couples having at least 5 common items and not having 10 or more common items is very low (due to the original nature of this data set).

It is evident that the LíbímSeTi data set distributions are quite different from the other three data sets. It appears that in this data set users pretty much agree on their opinions and that could be the main reason why collaborative filtering techniques are predetermined to be more than suitable for recommendations in the dating services environment.

The figure 2.5 shows the mean sizes of overlaps of ratings vectors for different similarities. The key point noticed are very low overlaps near similarity 1 for all data sets.

Scores Similarity Frequencies

The scores similarity frequencies, in the figure 2.6, display the distributions of similarities among users' scores vectors (the columns of the ratings matrix). The Jester data set distributions are not very informative as the data set contains only 100 valid columns ($\sim$ 100 very dense vectors, where each couple has just too many common items). All the other diagrams encourage the observation made in the previous paragraphs, although the differences are not so intense.

The adjusted scores similarities frequencies, as shown in the figure 2.8, display the distributions of similarities among user's scores vectors (the columns of the ratings matrix) adjusted by each user's ratings mean. This similarity metric is used in the adjusted Item-Item algorithm (see the section 3.4 on the page 19).

The figures 2.7 and 2.9 show the mean sizes of overlaps of scores vectors for different similarities, for both classic Pearson's correlation coefficient and its adjusted variant.

[6] Vector similarities - see the section 3.3.1 on the page 19

[7] To calculate the similarity of two given vectors, a certain minimum number of common (non-empty in both vectors) items is required. If the two vectors have less common items than such minimum, their similarity is considered undefined.

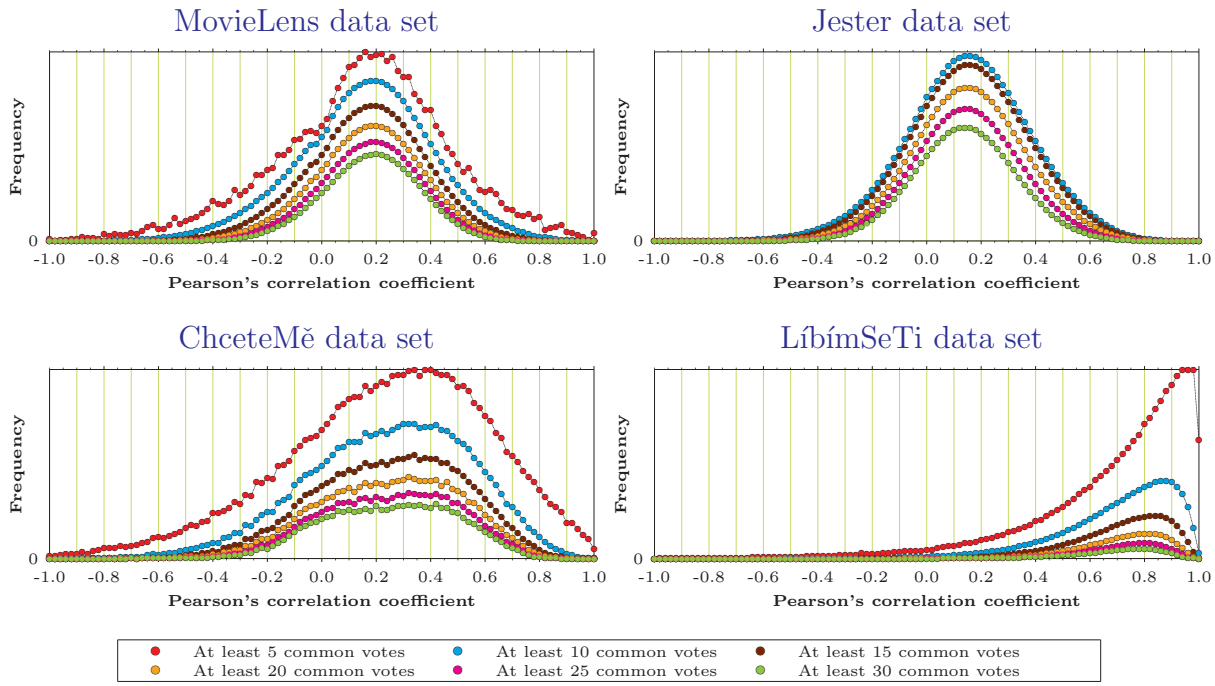


Figure 2.4: Distributions of rating similarities

The distributions of similarities among users' ratings vectors (the rows of the ratings matrix) for different limits on the minimal number of common votes in the vectors. The classic Pearson's correlation coefficient is used as the similarity metric.

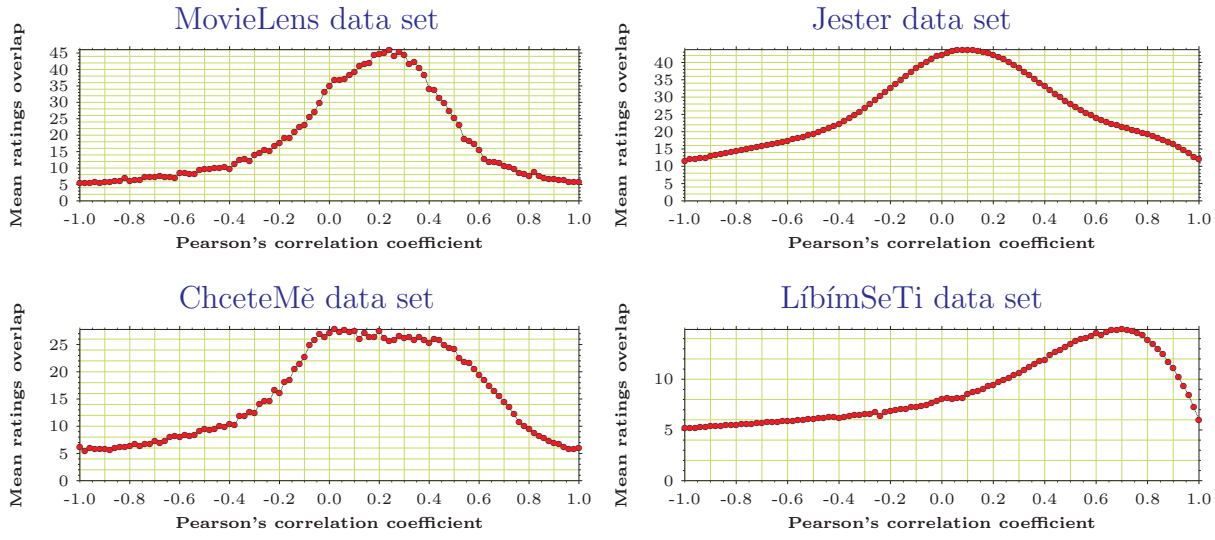


Figure 2.5: Mean sizes of rating overlap for different similarities

The distributions of ratings vectors' mean overlap sizes for different ratings vectors' similarities (among the rows of the ratings matrix). The classic Pearson's correlation coefficient is used as the similarity metric.

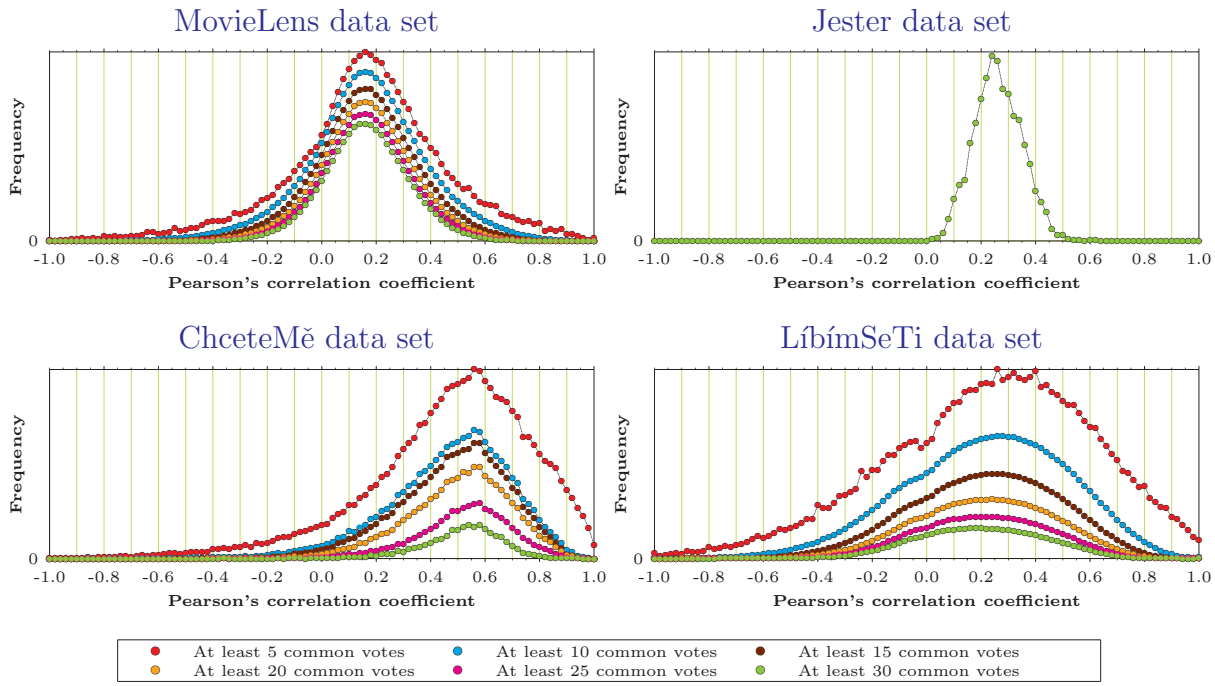


Figure 2.6: Distributions of score similarities

The distributions of similarities among users' scores vectors (the columns of the ratings matrix) for different limits on the minimal number of common votes in the vectors. The classic Pearson's correlation coefficient is used as the similarity metric.

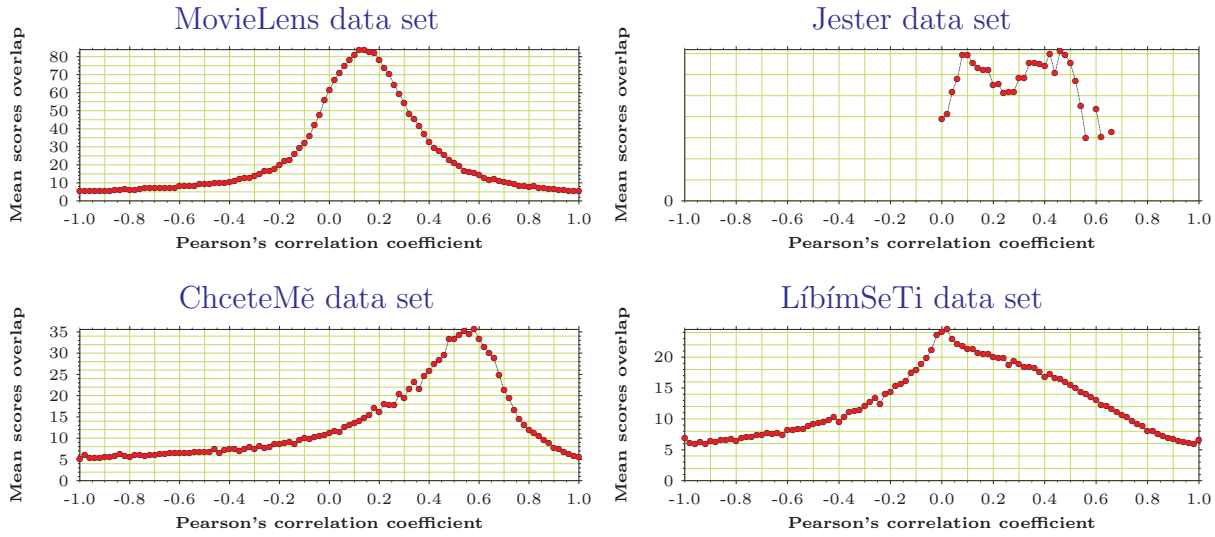


Figure 2.7: Mean sizes of score overlap for different similarities

The distributions of scores vectors' mean overlap sizes for different scores vectors' similarities (among the columns of the ratings matrix). The classic Pearson's correlation coefficient is used as the similarity metric.

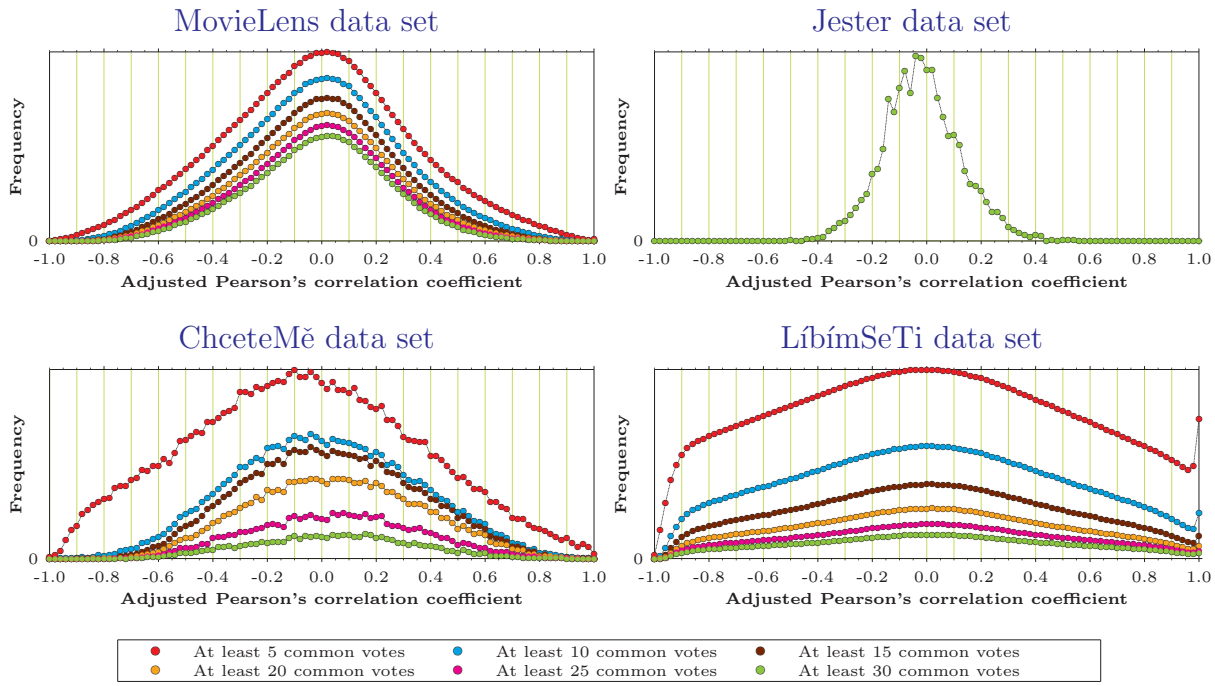


Figure 2.8: Distributions of score adjusted similarities

The distributions of similarities among users' scores vectors (the columns of the ratings matrix) for different limits on the minimal number of common votes in the vectors. The adjusted Pearson's correlation coefficient is used as the similarity metric.

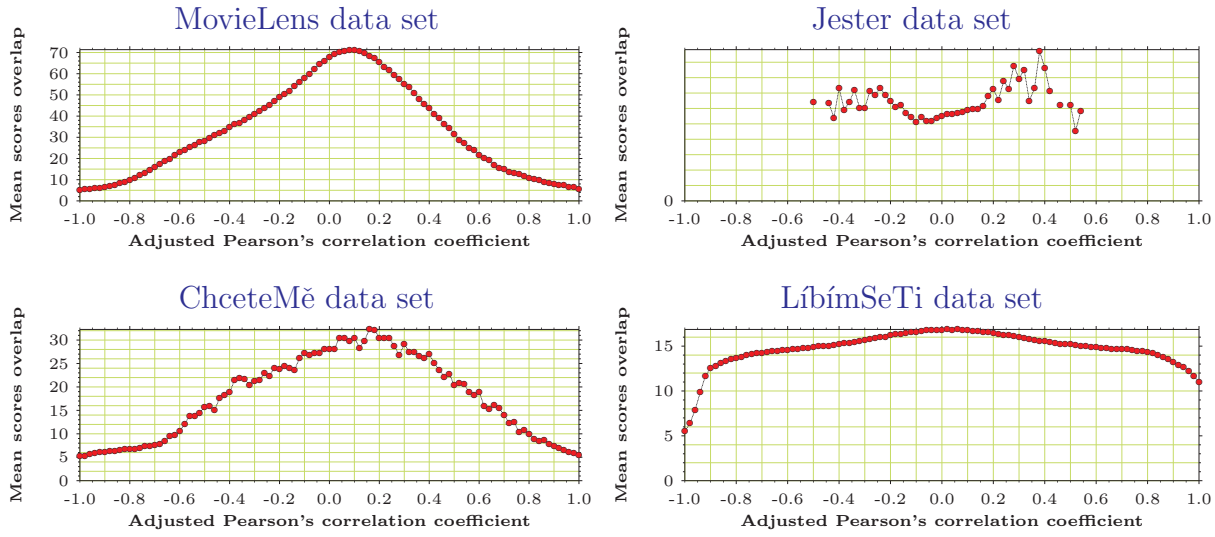


Figure 2.9: Mean sizes of rating overlap for different adjusted similarities

The distributions of scores vectors' mean overlap sizes for different scores vectors' similarities (among the columns of the ratings matrix). The adjusted Pearson's correlation coefficient is used as the similarity metric.

Chapter 3

Algorithms

This chapter focuses on the algorithms presented in this thesis and their main ideas.

The fundamental task in collaborative filtering is to predict a set of missing ratings of a particular user (the active user) based on the data contained in the ratings matrix (sometimes referred to as the user database). There exists the following two general approaches to collaborative filtering:

1. **Memory-based algorithms** – these algorithms operate in the online manner over the entire ratings matrix to make predictions
2. **Model-based algorithms** – these algorithms, in contrast, use the ratings matrix in the offline manner to estimate a model, which is then used for predictions

The memory-based algorithms usually yield better absolute results and are more adaptable to dynamic environments where the ratings matrix gets modified very often. On the other hand, the memory-based approach is very computationally expensive, so applications using large data sets may prefer the approximation offered by model-based algorithms.

This thesis examines two most popular collaborative filtering algorithms based on the (k)-Nearest Neighbours Algorithm: **User-User Nearest Neighbours Algorithm** and **Item-Item Nearest Neighbours Algorithm**. These are probably the most important algorithms in the memory-based category. Two more trivial algorithms (the **Random Algorithm** and the **Mean Algorithm**) are introduced to have a simple common base for later comparison.

3.1 Random Algorithm

According to the above "definitions", the Random Algorithm is more of a model-based then a memory-based collaborative filtering algorithm. Its every prediction is an uniformly distributed random value within the ratings value scale. For fixed users a and j it shall always predict the same random rating value $p_{a,j}$. Keeping this constraint ensures that

the algorithm's predictions stay random, but the algorithm's behaviour is invariable even when involved in the independent runs.

This algorithm with its theoretical possible accuracy results are mentioned in the Appendix A of [18].

3.2 Mean Algorithm

The Mean Algorithm is a very simple but surprisingly effective memory-based collaborative filtering algorithm. It is sometimes referred to as the Item Average Algorithm or the POP algorithm [9] as it falls within the so called *popularity prediction techniques* [37]. The prediction $p_{a,j}$ is calculated as the mean value of all the non-empty ratings targeted to the user j ($\sim$ the mean score for the user j). If the S_j is a set of users which have rated the user j, then the mean score for the user j is defined in (3.1) as:

$$\overline{s_j} = \frac{1}{|S_j|} \sum_{v \in S_j} r_{v,j} \tag{3.1}$$

and that is also the prediction ($p_{a,j} = \overline{s_j}$). It is obvious that this approach completely ignores the information entered to the system by the active user.

This algorithm together with the Random Algorithm act mainly as a baseline for an accuracy comparison among other presented algorithms and their variants.

3.3 User-User Nearest Neighbours Algorithm

The User-User variant of the (k)-Nearest Neighbours Algorithm is the most essential algorithm in the whole concept of collaborative filtering. The main idea of this approach is quite simple, yet ingenious. When predicting ratings of the active user a, the user database is first searched for users with similar ratings vectors to the user a (users with similar opinions or "taste" - the so called **neighbours**). The ratings similarity, in most of the literature denoted as $w(i, a)$, can reflect for example the distance or correlation between each user i and the active user a. The opinions of the k most similar neighbours are then used to form the predictions of the active user.

The predicted rating $p_{a,j}$ of the active user a for another user j is assumed to be a weighted sum of the ratings for user j of those k most similar neighbours, as shown in the following equation (3.2):

$$p_{a,j} = \overline{r_a} + \xi \sum_{i=1}^{k} w(a, n_i)(r_{n_i,j} - \overline{r_{n_i}}) \tag{3.2}$$

where n_i is the i-th nearest neighbour with a non-zero similarity to active user a and a non-empty rating $r_{n_i,j}$ for user j, ξ is a normalizing factor such that the absolute values

of the similarities $w(a, n_i)$ sum to unity, and $\overline{r_u}$ is the mean rating of the user u based on all his non-empty ratings. If the R_u is a set of users which the user u has rated, then the mean rating of the user u is defined in (3.3) as:

$$\overline{r_u} = \frac{1}{|R_u|} \sum_{v \in R_u} r_{u,v} \tag{3.3}$$

Various collaborative filtering algorithms based on the nearest neighbours approach are distinguished in terms of the details of the similarity calculation. This thesis concentrates on the similarity metric based on the well known Pearson's correlation coefficient, described in the following section.

3.3.1 Pearson's Correlation Coefficient

The **Pearson's product-moment correlation coefficient** or simply the *sample correlation coefficient*, is a measure of extent to which two samples are linearly related. This general formulation of statistical collaborative filtering first appeared in the published literature in the context of the GroupLens Research Project, where the Pearson's correlation coefficient was used as the basis for the similarity weights [31].

The correlation between the ratings of users a and j is shown in the equation (3.4):

$$w(a, j) = \frac{\sum_i (r_{a,i} - \overline{r_a})(r_{j,i} - \overline{r_j})}{\sqrt{\sum_i (r_{a,i} - \overline{r_a})^2 \sum_i (r_{j,i} - \overline{r_j})^2}} \tag{3.4}$$

where the summations over i are over the users which both users a and j have rated ($\sim$ common votes). For very small ratings vector overlaps, this similarity metric returns inaccurate results. That is why the actual implementations use a required minimum number of common votes to actually calculate a valid similarity[1].

3.4 Item-Item Nearest Neighbours Algorithm

The Item-Item variant of the (k)-Nearest Neighbours Algorithm uses a different point of view on the ratings matrix. Instead of utilizing the ratings matrix rows similarities (like the User-User variant), it utilizes the ratings matrix columns similarities. In the dating service environment, both rows and columns represent users, so the "Item-Item" qualifier adopted from the classic collaborative filtering scenarios might be a little misleading.

When predicting rating $p_{a,j}$ of the active user a, the user database is first searched for users with similar scores vectors to the user j (users being rated alike - also called neighbours). The users' scores similarity $\tilde{w}(i, j)$ is calculated in the same manner as for

[1] There exists another way of overcoming the "small ratings vector overlaps" problem called *default voting* [11].

the User-User variant between each user i and the target user j (only working with columns instead of rows of the ratings matrix). The ratings of the active user a for the k most similar neighbours to j are then used to form the prediction $p_{a,j}$, as shown in (3.5):

$$p_{a,j} = \overline{s_j} + \xi \sum_{i=1}^{k} \tilde{w}(j, n_i)(r_{a,n_i} - \overline{s_{n_i}}) \tag{3.5}$$

where n_i is the i-th nearest neighbour with a non-zero similarity to the target user j and a non-empty rating r_{a,n_i} from user a, ξ is a normalizing factor such that the absolute values of the similarities $\tilde{w}(j, n_i)$ sum to unity, and $\overline{s_u}$ is the mean score for the user u, as explained in (3.1).

3.4.1 Adjusted Pearson's Correlation Coefficient

One major difference between the similarity computation in the User-User variant and the Item-Item variant of the (k)-Nearest Neighbours Algorithm is that in case of the User-User variant the similarities are computed along the rows of the ratings matrix but in case of the Item-Item variant the similarities are computed along the columns of the ratings matrix. The similarity in case of the Item-Item variant suffers from the fact that each pair of common votes during the similarity calculation belongs to a different user with different subjective ratings scale. The adjusted Pearson's correlation coefficient is trying to answer this drawback by subtracting the user's mean rating from each such pair. Formally, the similarity of scores of users j and l is described in (3.6):

$$\tilde{w}_{adj}(j, l) = \frac{\sum_i (r_{i,j} - \overline{r_i})(r_{i,l} - \overline{r_i})}{\sqrt{\sum_i (r_{i,j} - \overline{r_i})^2 \sum_i (r_{i,l} - \overline{r_i})^2}} \tag{3.6}$$

where the summations over i are over the users which have rated both users j and l ($\sim$ common votes).

Chapter 4

ColFi Architecture

This chapter outlines the ColFi System architecture design and the main decision steps leading to it.

4.1 ColFi Architecture Overview

ColFi System architecture is designed to be maximally simple and flexible, so users or developers can focus on collaborative filtering instead of the system complexity. The core of the system is the server side of typical stateless TCP/IP client-server solution. It consists of a global data manager, a set of ColFi services and a communication module with its own specific communication protocol (**CCP** - ColFi Communication Protocol). The data manager serves as the main data provider for all the components in the system and ColFi services contain logic exposed to the outside world (to the clients). Particular data manager and ColFi services implementations are provided as server plug-ins.

The illustration 4.1 summarizes ColFi architecture design:

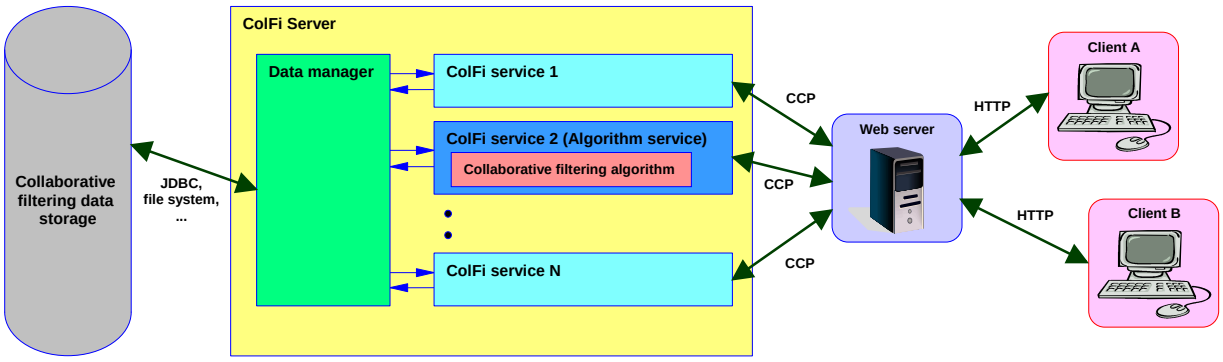


Figure 4.1: ColFi architecture

The communication protocol between the data storage and the data manager is implementation dependent. Each ColFi service is represented by one TCP/IP port number.

4.2 Pure Client-Server Solution

Pure stateless TCP/IP client-server solution was chosen among many others for its simplicity, platform independency, scalability and mainly for the best performance results. Each ColFi service instance has its own unique TCP/IP port number and is independent on all the other ColFi services. Thus, multiple client connections are supported and handled in parallel.

ColFi Server contains a communication module with its own specific client-server communication protocol (CCP), which is, in a way, very similar to popular Java RMI[1] protocol. Again, CCP is a very simple protocol and it allows only limited remote method invocations. That makes it very fast (unlike SOAP[2]) and does not require anything special on the client side (unlike Java RMI). See the section 5.1.5 on the page 28 for details.

Number of other solutions were implemented and rejected, either for weak overall performance (e.g., Web Services) or for complicated installation and setup of the server environment for end users of the ColFi System (e.g., EJB, JSP, Servlets).

4.3 Data Manager

The data manager is the heart of the ColFi Server. It is a general interface that provides all the other ColFi components with data necessary for collaborative filtering. There is always exactly one data manager instance per server instance. Its concrete implementation is in a form of a plug-in to the ColFi Server and its actual behavior can differ in many ways. The data manager can cache data in the memory or it can directly access some other data storage (e.g., database or specifically designed file system). In some cases, read only implementation may be desired and so on.

Although both cached and direct access is supported, only cached variants achieve reasonable performance parameters. Direct access is meant to be used mainly with small data sets or in some specific test scenarios.

4.4 ColFi Services

The ColFi service is the only part of the system exposed to the outside world. Each service has its own TCP/IP server port number where it listens for incoming client requests. ColFi services are stateless (sessionless), so each request is both atomic and isolated and should not affect any other instant requests (that allows simple parallel handling of multiple client connections from the server's point of view). The only requirement to each ColFi service is that it should be able to get associated with the data manager instance.

[1] RMI (Remote Method Invocation) – remote method invocation protocol implemented in Java, which forces the client to use Java

[2] SOAP (Simple Object Access Protocol) – general and platform independent remote object access protocol based on XML, which makes it quite slow

The ColFi service is not limited to do collaborative filtering, it is only limited to do operations with data obtained from the data manager (it can do some statistical operations for example).

Implementation details are discussed later in the section 5.3 on the page 34.

4.5 Algorithm Service

The algorithm service is special abstract implementation of the ColFi service with focus on collaborative filtering. It contains an Algorithm interface, which represents collaborative filtering algorithm. By default, this ColFi service exposes basic matrix in/out data operations together with prediction and recommendation methods.

The detailed implementation is described in the section 5.3.1 on the page 35.

Chapter 5

ColFi Implementation

This chapter focuses on the implementation details of the ColFi System and is intended to be read mainly by developers.

The main implementation goals were simplicity, good overall performance, enough abstraction, system extensibility and target platform independency. To achieve these goals, the J2SE™ 5.0[1] was chosen for its good abstraction, platform independency and reasonable performance issues. The ColFi System is implemented to run in the latest version[2] of the Sun Microsystems JVM which is the most popular and widely used Java solution. The presented implementation is currently tested in practice as a part of real world web application at http://www.libimseti.cz/ (running the Gentoo Linux distribution[3]).

Java source files are compiled into four separate `jar` files for better transparency and development administration:

- `colfi-server.jar` – the main ColFi Server implementation classes, network communication, threads management
- `colfi-commons.jar` – public common utilities and handy routines, fast arrays, simplified hash table, general configuration and logging facilities
- `colfi-datamanagers.jar` – the data manager implementations, possibly dependent on some 3^{rd} party libraries (e.g., database drivers)
- `colfi-algorithms.jar` – the algorithm service and all the algorithm implementations ($\sim$ collaborative filtering and recommender core)

As `colfi-server.jar` contains the Java main method, it acts as the main "executable" part of the system. The rest of the `jar` files are understood as plug-ins, from which only `colfi-commons.jar` is required to successfully start the standalone ColFi Server.

[1] J2SE™ 5.0 (Java™ 2 Platform Standard Edition 5.0) – http://java.sun.com/

[2] Currently J2SE™Development Kit 5.0 Update 7.

[3] Gentoo Linux – http://www.gentoo.org/

The details not mentioned in this chapter can be found in the form of a standard Javadoc[4] documentation as a part of the ColFi Project.

5.1 ColFi Server

As stated above, ColFi Server implementation classes are stored in `colfi-server.jar`. The Java main method is located in the class `Server` and it does the following:

- Initializes logging – every server run is logged into a separate log file with unique timestamp in its name
- Initializes configuration – opens the configuration property file based on the server's command line argument (see further)
- Initializes data manager – creates concrete data manager instance based on the current configuration and initializes it
- Initializes ColFi services – creates concrete service instances based on the current configuration and initializes them (passing them the global data manager instance created earlier)

5.1.1 Logging

ColFi Server logging facilities are based on standard Java logging API, which was chosen instead of other popular logging libraries (like Log4j[5]) for its simpler configuration and sufficient features. Also no other 3^{rd} party library is necessary this way, because it is all part of J2SE. ColFi Server logs its activity into standard output console and into XML[6] log file `log/colfi-[X].log`, where `X` stands for the timestamp of server start.

This default behavior is defined in the class `Logger` and it may be extended by a developer. Standard way of logging things from inside the system is done through the singleton pattern as follows:

```
Logger.getInstance().log(Level.WARNING, "Cannot close file \"" + fName + "\".", ioException);
```

5.1.2 Configuration

ColFi Server configuration is stored in a standard property file, which is specified by the server's command line argument. Only the first command line argument `arg1` is used

[4] Javadoc is a tool for generating API documentation in HTML format from doc comments in source code.

[5] Log4j – part of the open source Apache Logging Services project, designed for Java environment (http://logging.apache.org/log4j/)

[6] Resulting XML's DTD is http://java.sun.com/dtd/logger.dtd.

to form the path to the property file as `conf/[arg1].properties`. When no command line argument is present, value `default` is used instead. The configuration file is required.

There are two basic techniques used to get to the configuration properties. The simpler of the two is just reading one property from the configuration file. For example this is the way to define global configuration property, whether ColFi Server should resolve client's IP addresses upon incoming requests:

```
colfi.resolveClientIP = true
```

And this is the standard way of reading such configuration property through the singleton pattern (static methods):

```
final boolean resolveClientIP = CommonTools.booleanValue(Configuration.getBoolean("colfi.resolveClientIP"));
```

This is the basic mechanism of reading one simple configuration property and using it in the system. The class `Configuration` contains methods for reading numbers, strings and boolean values.

There also exists a second, more complex, approach. It is used for universal instantiation of the classes which are unknown till runtime, through Java reflexion. This mechanism actually substitutes general java class constructor calls. Let `keyPrefix` be the configuration key for such class. Then the property `[keyPrefix].class` contains a fully qualified java class name of the desired class. Such class must implement `Configurable` interface and must have a public constructor that takes no parameters. The class instance is then obtained as follows:

- Constructor with no parameters is called.
- Class is scanned for methods with `ConfigurableParameter` annotation[7] present. Each of these methods correspond to some value in the configuration file and the configuration key for that value is determined from `parameterName` field of the annotation (`[keyPrefix].parameter.[parameterName]`). There are two more annotation fields considered before the actual method call is invoked: `required` - whether this parameter is required in the configuration file; and `defaultValue` - default value used when no value is found in the configuration file. If all looks good, the method is invoked with obtained value as an argument (automatic type conversion is applied). Each annotated method is supposed to have exactly one argument and its type is limited to some primitive types (`boolean`, `byte`, `int`, `long`, `String`) or it can be another `Configurable` object (yes - it can go recursive in here).
- After all the annotated methods are called, method `configurationDone()` from the `Configurable` interface is called and the final instance is considered ready for use.

This is the actual example from the configuration file, where `FileDataManager` class should serve as the main ColFi Server data manager instance:

[7] See http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

```
colfi.dataManager.class = cz.cuni.mff.colfi.data.cached.file.FileDataManager
colfi.dataManager.parameter.fileName = /tmp/colfi-tmp-matrix.ser
colfi.dataManager.parameter.value.no = 0
colfi.dataManager.parameter.value.min = 1
colfi.dataManager.parameter.value.max = 10
```

One of the annotated methods[8] in the `FileDataManager` implementation looks like this:

```
@ConfigurableParameter(parameterName = "fileName", required = true)
public final void setFileName(final String newFileName) {
    fileName = newFileName;
}
```

And this is the standard way of instantiating such class in the system:

```
final DataManager dataManager = (DataManager)Configuration.getConfigurableInstance("colfi.dataManager");
```

5.1.3 Data Manager Instance

Even though data manager is the heart of the ColFi Server, there is no assumption to its implementation other then that it is supposed to implement the general `DataManager` interface (see further). The `DataManager` interface extends the `Configurable` interface, therefore the specific data manager instance is obtained through configuration file as described above. The configuration property key is `colfi.dataManager`.

There is always exactly one data manager instance per server instance. It is global for all the ColFi Server components ($\sim$ ColFi services).

5.1.4 ColFi Service Instances

ColFi services are instantiated in the same manner as the data manager. Again, the only assumption about service is that it implements the `Service` interface which also extends the `Configurable` interface. Specific service instances are obtained through configuration file. Their configuration property keys are `colfi.service.[n]`, where `n` goes from 1 to the total number of services. There is no limit on the total number of ColFi services per server instance. For each service there are two more configuration property keys (for server purposes): `colfi.service.[n].name` - contains the name of the service, and `colfi.service.[n].port` - contains the server port on which that service should listen for incoming requests.

Each ColFi service gets initialized with the global data manager created earlier and then gets started in its own thread which is implemented in the class `ServiceThread`.

Each incoming connection (possibly containing multiple requests) to the ColFi service is served in an extra new thread which is implemented in the class `ResponseThread`. That is also the place, where the ColFi Communication Protocol is mainly implemented.

[8] See the complete `FileDataManager` source code for the rest of the annotations.

5.1.5 ColFi Communication Protocol

ColFi Server uses its own communication protocol. Every client request is basically a remote ColFi service method invocation and server response is serialized return value of that method call. Service method may have some arguments of these primitive types: `byte`, `int` or `boolean`. This limitation can be easily extended by a developer, although there is no need for other argument types. Client request consists of the ColFi service method name (which is case sensitive) followed by left parenthesis, *argument types definition*, which is closed by right parenthesis and after that there are serialized method arguments (big-endian representation). The argument types definition is formed in the following manner:

- Each character represents one argument type.
- The definition is case sensitive: `b` stands for `boolean`, `B` for `byte` and `I` for `int`.

This is an example client request for invoking ColFi service method `getRating` with two `int` arguments (actual values used are 1 and 12345):

g	e	t	R	a	t	i	n	g	(	I	I	)				1				12345	
0x67	0x65	0x74	0x52	0x61	0x74	0x69	0x6E	0x67	0x28	0x49	0x49	0x29	0x00	0x00	0x00	0x01	0x00	0x00	0x30	0x39	

The response of the server consists of the response error code (one byte) and the result. The response error code value has these meanings:

- `0x00` – OK, no error occurred and valid result follows (when not void)
- `0x01` – unknown command (unknown ColFi service method)
- `0x02` – unexpected end of a input stream from the client side
- `0x03` – some ColFi internal service exception

Generally, non zero value means some error has occurred. In that case, serialized `String` with exception description follows. In case of no error, serialized return value of invoked method is returned. The client must know the form of the result content to be able to parse it. Primitive types are just simply written into the output stream (again big-endian representation) and other objects must implement `StreamableToByteArray` interface to be successfully "serialized" into the output stream. This could be also very easily extended by a developer if this should not suffice.

5.2 Data Managers

As mentioned above, ColFi Server includes exactly one global data manager instance which is supposed to implement the `DataManager` interface. The role of this data manager instance is to provide access to the collaborative filtering data to all the other components in

the system (mainly to all the ColFi services). The data being provided is the content of the ratings matrix together with the list of so called active users. Users are identified by their unique id (32 bits) and the data manager should know whether a user with given id exists (is active) or not. All the matrix values on the server side are converted to the internal value scale as defined in the `DataManager` interface. Only the `updateRating(int,int,byte)` method uses external value scale for its last argument, because the value it gets origins from the client side. Both internal and external values are limited to 8 bits. The last global contract to the implementation of the `DataManager` interface is that self-ratings are not allowed. In other words, it is not possible for any user to rate himself and therefore the main diagonal of the ratings matrix should always be empty.

The figure 5.1 shows the complete list of the methods in the `DataManager` interface.

```
public interface DataManager extends Configurable {
    /** Returns data manager's name. */
    String getName();
    /** Returns data manager's implementation version. */
    byte getVersion();
    /** Returns 'null' value of external scale. */
    byte getExternalNoValue();
    /** Returns minimum value of external scale. */
    byte getExternalMinValue();
    /** Returns maximum value in external scale. */
    byte getExternalMaxValue();
    /** Closes this data manager. No further operations are allowed. */
    void close() throws DataManagerException;
    /** Gets the lock which should be used for multi-read operations to preserve their isolation. */
    Lock getReadLock();
    /** Adds a new listener for ratings matrix modification events. */
    void addDataManagerListener(DataManagerListener l);
    /** Removes attached listener for ratings matrix modification events. */
    void removeDataManagerListener(DataManagerListener l);
    /** Returns all active users sorted by ids in ascending order. */
    SortedAscIntArray getAllUsers() throws DataManagerException;
    /** Returns users not rated by user u1. Unsorted. */
    IntArray getNotRatedUsers(int u1) throws DataManagerException;
    /** Returns users who have not yet rated user u2. Unsorted. */
    IntArray getNotScoredUsers(int u2) throws DataManagerException;
    /** Adds new user to ratings matrix. It gets next available id, which is then returned. */
    int addUser() throws DataManagerException;
    /** Adds new user to ratings matrix with specified id. Throws exception when given id is already occupied. */
    void addUser(int u) throws DataManagerException;
    /** Removes existing user from ratings matrix. */
    void removeUser(int u) throws DataManagerException;
    /** Returns rating r_{u1,u2}. */
    byte getRating(int u1, int u2) throws DataManagerException;
    /** Returns all ratings from user u1 sorted by user id in ascending order. */
    SortedAscIntFixedByteArray getRatings(int u1) throws DataManagerException;
    /** Returns all scores for user u2 sorted by user id in ascending order. */
    SortedAscIntFixedByteArray getScores(int u2) throws DataManagerException;
    /** Updates rating r_{u1,u2}. External 'null' value can be used to delete specified rating. */
    void updateRating(int u1, int u2, byte externalValue) throws DataManagerException;
}
```

Figure 5.1: `DataManager` interface

The `DataManager` interface, which represents the main data storage access point for all the other components in the ColFi System.

All data manager implementations have to be careful about the fact that their instance is used in multi-threaded environment. So if it is intended to be used with more than one ColFi service (or more precisely in a multi-client environment), the implementation

should be thread-safe[9]. That might be particularly difficult when dealing with modifiable implementation (meaning that one of the methods `addUser`, `removeUser` or `updateRating` is implemented). For such modifiable implementations there is prepared standard java listener pattern (class `DataManagerListener`) for other components using the data manager to get notification events about the data manager content modifications. The figure 5.2 shows the complete list of the methods in the `DataManagerListener` interface.

```
public interface DataManagerListener extends EventListener {
    /** User u was added. */
    void userAdded(int u);
    /** User u was removed. */
    void userRemoved(int u);
    /** New rating r_{u1,u2} with value value was added. */
    void ratingAdded(int u1, int u2, byte value);
    /** Existing rating r_{u1,u2} was modified from value oldValue to value newValue (oldValue != newValue). */
    void ratingUpdated(int u1, int u2, byte oldValue, byte newValue);
    /** Existing rating r_{u1,u2} was removed. Previous value was oldValue. */
    void ratingRemoved(int u1, int u2, byte oldValue);
}
```

Figure 5.2: `DataManagerListener` interface

The `DataManagerListener` interface, which represents the standard java event listener, able to receive the data manager modification events.

The meaning of these event listener methods is obvious. All the "value" arguments used are in the ColFi's internal value scale. Although the `DataManager` interface contains only one method for actual ratings matrix modification (`updateRating`), there are three separate methods in the `DataManagerListener` for ratings matrix modification events (`ratingAdded`, `ratingUpdated` and `ratingRemoved`). This feature might make data manager implementation little more complicated, but on the other hand it also helps a lot in implementing the modification event consumers (see the section 5.4.2 on the page 36).

There are methods in the `DataManager` interface that are actually redundant and their return value could be acquired with the help of other data manager methods. One such method is `getNotRatedUsers`. The reason to have these methods in the interface is the possibility that some implementations may have data stored in a very specific way and could return such values directly improving overall performance. The default implementation of these methods, based on other `DataManager` methods is written in the `DataManagerAdapter` class. This class also implements the listener pattern together with some other useful methods for eventual descendants.

5.2.1 Cached Matrix and Cached Data Manager Adapter

All the presented cached data manager implementations are descendants of the class `CachedDataManagerAdapter`. It is an abstract implementation of modifiable data manager, which caches all the data in the memory in the data structure called **cached matrix**. It implements all the essential `DataManager` interface methods and the only thing the

[9]Thread-safe code is code that will work even if many threads are executing it simultaneously.

descendant class must take care of is the initial filling of cached matrix by implementing the `initCachedMatrix` method. The implementation is thread-safe.

The cached matrix structure is implemented in the `CachedMatrix` class. It is a complex data structure representing the entire ratings matrix content. The matrix is stored as a map[10] of rows (hashed by user ids) and as a map of columns (again hashed by user ids). This "two-way" storage uses twice the space but speeds up most of the desired operations over the matrix. The relevance of particular user id is equivalent with the existence of non-null entries in both maps for such user id.

Each row and column is stored as a list of pairs (user id and value) sorted by user id in ascending order. This particular list is implemented in `SortedAscIntFixedByteArray` as a part of the `arrays` (`arrays.multi`) package in `colfi-commons.jar`. This package contains various forms of array implementations designed for maximum performance. There are implementations of modifiable arrays of few primitive types and all of them share common effort to minimize the number of memory reallocations during their "life". The main idea behind their implementation is to allocate some constant space in advance so later array modifications can be mostly handled without the time consuming memory reallocations. This feature uses a little extra memory, but it significantly improves performance. Also, such array instances are often passed as method arguments inside the ColFi System and they get rarely duplicated. The original object is passed as it is, so developers have to be careful about unintentional modifications.

Cached matrix implementation is fully synchronized so in other words thread-safe. The illustration 5.3 recapitulates preceding paragraphs.

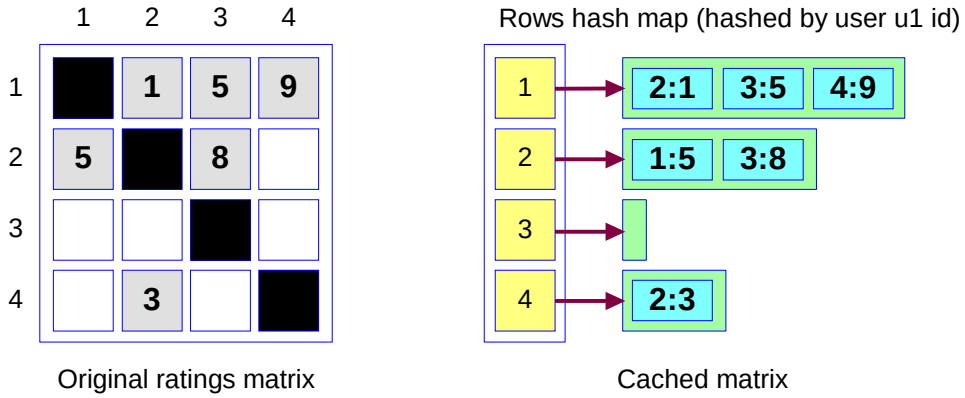


Figure 5.3: Cached matrix implementation

An example ratings matrix and its representation in memory as a cached matrix data structure. Only rows hash map is demonstrated as the columns hash map is similar. The empty array hashed under user 3 denotes that this user exists but has no ratings in the ratings matrix.

[10]Own special `IntHashMap` is used instead of the standard java java.util.Map interface. This implementation allows only primitive int keys which makes it about 30% faster and consumes less memory.

Because the cached matrix structure is the performance bottleneck for all the presented data manager implementations, it is important to know its memory and time complexity.

The memory complexity is affected by the fact that the matrix is stored both as a map of rows and a map of columns. The memory used is proportional to twice the number of ratings in the matrix. The memory complexity is therefore linear with the number of ratings. To observer some real numbers, current cached matrix implementation uses about 74 MB of memory to store matrix with 100,000 users and 5,000,000 random ratings.

The time complexity is much more important characteristics. The mostly used operation over the cached matrix is reading rows, columns or explicit ratings. Getting rows or columns can be done with a constant time complexity, because it is just looking up the user id key in one of the two hash maps. Getting a single explicit rating is little more complicated. Let N be the total number of active users and let R be the total number of ratings in the matrix. In average case, ratings are uniformly distributed over the whole matrix and the matrix itself is very sparse ($N^2 \gg R \gg N$). The row and also the column of the target rating is obtained in constant time. Both are arrays of pairs sorted by user id, so either one can be used to perform binary search for desired target rating. Assuming the uniform ratings distribution, it does not matter which one is used for the search (although of course real implementation do choose the smaller one). The average length of such an array is R/N, so the time complexity to get an explicit rating is $O(\log R/N)$.

The important fact is that getting rows and columns is done with a constant time complexity and that it actually returns the array sorted by user id. That is greatly utilized later in the algorithm implementations (see section 5.4.3 on page 37 for details).

5.2.2 Empty Data Manager

The simplest possible descendant of the `CachedDataManagerAdapter` is implemented in the class `EmptyDataManager`. It initializes the cached matrix with no data and no users. It is fully functional modifiable data manager, it just starts with an empty initial ratings matrix. It is mainly used for automatic testing of the `CachedDataManagerAdapter` class and the data manager concept altogether. Another application could be in situations where ratings matrix persistence is not important.

5.2.3 File Data Manager

Because every `DataManagerAdapter` descendant must implement a way of serializing its data into the file (method `serializeDataIntoFile`), there should exist a data manager capable of reading those serialized files. The class `FileDataManager` is such a data manager implementation based on the `CachedDataManagerAdapter`. It initializes its cached matrix with data from the serialized file and it expects the file to contain plain serialized `CachedMatrix` java object instance. By a convention, the standard file name extension for serialized java objects is *ser*.

And for example, the total size of serialized cached matrix with 100,000 users and 5,000,000 random ratings is about 74 MB (both rows and columns map gets serialized into

the file because it makes deserialization of the data about 6 times faster, even though the file uses twice the size).

5.2.4 Cached MySQL Data Manager

The majority of ordinary applications do store their ratings matrices in some persistent way. Mostly in databases. The class `CachedMySQLDataManagerAdapter` extends `CachedDataManagerAdapter` and implements data manager based on the MySQL[11] database. This adapter class has two descendants: `CachedMySQLDataManager` and read-only version `CachedMySQLReadOnlyDataManager`. Although they are both modifiable data manager implementations, only the former reflects the modifications back to the database.

Even though specific database engine was chosen, the implementation is not really dependent on it as it uses JDBC[12] technology. It is commonly used standard for Java applications to connect to the database. Databases of all major vendors do have implemented their JDBC drivers so they can be easily accessed from applications in an uniform way.

The ratings matrix data are loaded from one database table which is required to have at least these three database columns: `user-id1`, `user-id2` and `value`. Each rating is stored as one database record in that table. Values out of external value scale and values on the main diagonal of the ratings matrix are silently ignored.

As MySQL database server drops client connections after 8 hours of inactivity, there is the `MySQLAutoAlivePingThread` class, which periodically touches the database to keep the connection alive.

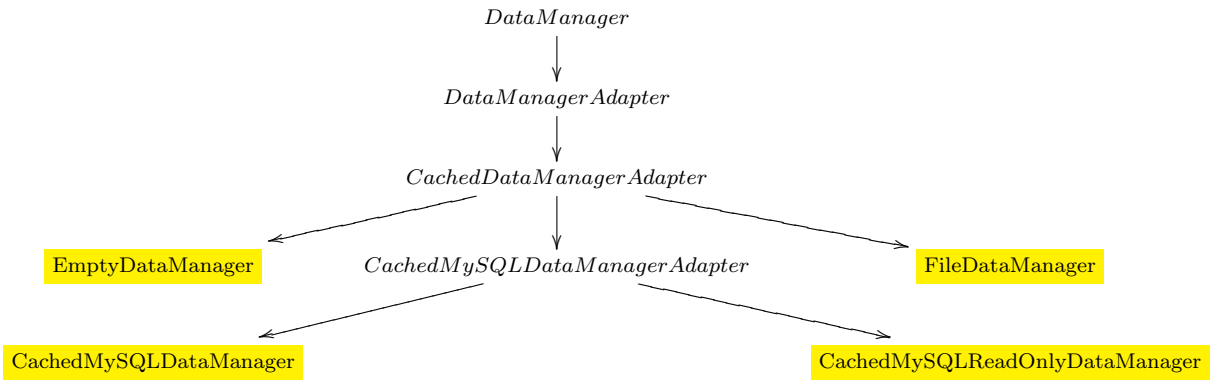


Figure 5.4: `DataManager` implementations hierarchy

This diagram shows the data manager classes hierarchy (highlighted classes are final data managers ready for use, the rest are interfaces or abstract adapter classes):

[11] MySQL (the world's most popular open source database) - http://www.mysql.com/
[12] JDBC - Java DataBase Connectivity

5.3 ColFi Services

ColFi service is an isolated server component exposed to the client side of the system. It contains the server "logic" over the data set provided by the global data manager instance. Each service is obliged to implement the `Service` interface (which also extends the `Configurable` interface, so it can be instantiated from the configuration property file). The `Service` interface has only one method, as shown in the figure 5.5.

```
public interface Service extends Configurable {
    /** Initializes this Service data source. */
    void setDataManager(DataManager newDataManager);
}
```

Figure 5.5: `Service` interface

The `Service` interface represents the ColFi service - a part of the ColFi System exposed to the outside world. The one thing required from the service is that it should be able to associate itself with a given data manager during its initialization phase.

This method is used during the service initialization phase to pass the global data manager instance to it. It is invoked exactly once, and it is not supported to later switch the data manager (although this could be extended by a developer if needed). And that is the only thing required from the service implementation.

Every ColFi service implementation has several methods, which are intended to be exposed to the clients. The `ServiceMethod` annotation is a method marker annotation used to specify such methods. The annotation has one boolean parameter named `quit` used to select the service "exit" method(s) - its invocation immediately terminates the service (the main ColFi Server exits when there are no more running ColFi services). As mentioned earlier, there exist some restrictions about service methods parameter and return types. Parameter types are limited to these primitive types: `boolean`, `byte` and `int`. Method return type is limited to these types: `boolean`, `Boolean`, `byte`, `Byte`, `int`, `Integer`, `String` and implementations of the `StreamableToByteArray` interface. The method is supposed to throw the `ServiceException` when anything goes wrong and the client should know about it. Any other exception is considered "unexpected". The following is an example service method from one service implementation:

```
@ServiceMethod
public final byte predictRating(final int u1, final int u2) throws ServiceException;
```

The `ServiceAdapter` class is an abstract `Service` implementation which exposes most of the data manager methods as service methods. That actually allows the client to work directly with the associated data manager. The complete list of exposed methods is:

```
String getDataManagerName();
byte getDataManagerVersion();
SortedAscIntArray getAllUsers() throws ServiceException;
IntArray getNotRatedUsers(int u1) throws ServiceException;
IntArray getNotScoredUsers(int u2) throws ServiceException;
int addUser() throws ServiceException;
void addUser(int u) throws ServiceException;
void removeUser(int u) throws ServiceException;
byte getRating(int u1, int u2) throws ServiceException;
SortedAscIntFixedByteArray getRatings(int u1) throws ServiceException;
SortedAscIntFixedByteArray getScores(int u2) throws ServiceException;
void updateRating(int u1, int u2, byte value) throws ServiceException;
```

5.3.1 Algorithm Service

The algorithm service is the main presented `Service` implementation based on the abstract `ServiceAdapter` class. It is a basic ColFi service for all collaborative filtering and recommender routines. The main collaborative filtering computational logic is contained in the service as an implementation of the `Algorithm` interface. The figure 5.6 shows the complete list of all the methods in that interface (methods marked with black arrow are exposed to the clients by the `AlgorithmService` class):

```
public interface Algorithm extends Configurable, DataManagerListener {
    /** Returns algorithm's name. */
  ▶ String getName();
    /** Returns algorithm's implementation version. */
  ▶ byte getVersion();
    /** Return associated data manager (possibly null). */
    DataManager getDataManager();
    /** Associates given data manager with this algorithm (null to detach current one). */
    void setDataManager(DataManager dataManager) throws AlgorithmException, DataManagerException;
    /** Predicts rating for r_{u1,u2}. */
  ▶ byte predictRating(int u1, int u2) throws AlgorithmException, DataManagerException;
    /** Predicts ratings for user u1. Using all users or just unrated ones. */
  ▶ FixedIntFixedByteArray predictRatings(int u1, boolean all) throws AlgorithmException, DataManagerException;
    /** Predicts scores for user u2. Using all users or just unscored ones. */
  ▶ FixedIntFixedByteArray predictScores(int u2, boolean all) throws AlgorithmException, DataManagerException;
    /** Recommends top-n users for the user u. Using all users or just unrated ones. */
  ▶ IntArray recommendUsers(int u, int n, boolean all) throws AlgorithmException, DataManagerException;
}
```

Figure 5.6: `Algorithm` interface

The `Algorithm` interface represents a collaborative filtering algorithm. It is used by an algorithm service. The boolean parameter named `all` tells the method to either use all the users in the data set or to use just users with no real value in the ratings matrix (unrated users).

The `Algorithm` implementation is associated with the data manager instance. Unlike the `Service` implementations, the `Algorithm` implementations are supposed to support online data managers switching (that is mainly for the research purposes - for example a learning algorithm of some kind might learn on one data set and then switch to another data set and run some tests).

Notice that the `Algorithm` interface also extends the `Configurable` interface, and therefore the actual instances are created and based on the configuration property file.

5.4 Algorithms

Every collaborative filtering algorithm implementation must realize the `Algorithm` interface mentioned in previous sections. Most implementations have some common characteristics. The `AlgorithmAdapter` abstract class assumes for its descendants that all predictions and recommendations are based on one method (`predictRating`), and that each such prediction is independent on any other predictions. There is also convention that if there exists real value in the ratings matrix for the rating $r_{u1,u2}$, then the prediction for this rating is that particular value. Under these assumptions, the `AlgorithmAdapter` implements all the other `Algorithm` interface methods and leaves just one method to its descendants:

```
byte predictRatingReal(int u1, int u2) throws AlgorithmException, DataManagerException;
```

This method is called when the actual prediction is needed, which means there is no real value in the ratings matrix for $r_{u1,u2}$. All `Algorithm` implementations based on `AlgorithmAdapter` have to implement the `predictRatingReal` method and also all the methods from the `DataManagerListener` interface as the `Algorithm` interface extends it and `AlgorithmAdapter` leaves their implementations to its descendants.

5.4.1 Random Algorithm

The Random algorithm (the `RandomAlgorithm` class) is the simplest descendant of the `AlgorithmAdapter` class. Its every prediction is a random value within the internal value scale. For fixed users `u1` and `u2` it always predicts for the rating $r_{u1,u2}$ the same random value. Each instance has its own set of random generator instances (see java.util.Random). All the data manager modification events are ignored as there is nothing cached inside the algorithm instance. This algorithm has no configurable parameters.

The `RandomAlgorithm` implementation is thread-safe and both time and memory complexity are constant.

5.4.2 Mean Algorithm

The Mean algorithm (the `MeanAlgorithm` class) calculates its prediction for the rating $r_{u1,u2}$ as the mean value of all the ratings for the user `u2` ($\sim$ the mean score of the user `u2`). The mere implementation is trivial, but there are two interesting principles.

All the array implementations from the `colfi.arrays` package do cache their total summation at all times, so the calculation of any array's mean value is in fact just one simple division. But this is dependent on the data manager implementation (for example direct data manager implementations still have to read the arrays from somewhere).

The second and more interesting thing is, that the `MeanAlgorithm` class may cache all the means it used itself. The mean is cached in the `CachedMean` class as a summation and a size of an array (so again getting the mean is a question of a simple division -

and that is done only once, because then it is cached in the `CachedMean` object as a `byte` return value to even avoid the division). The `MeanAlgorithm` implements all the `DataManagerListener` methods and corrects the appropriate cached means every time a data manager modification event arrives - so it does not need, possibly slow, data manager call to calculate the mean again. Of course this is just a simple example of utilization of the data manager listener callbacks and is really contra productive when using together with a cached data manager (due to many `CachedMean` objects allocations).

The Mean algorithm has one boolean configurable parameter called `useCaching`, used to turn on/off the mean-caching as described above. For the reasons mentioned, direct data managers should use the mean caching and cached data managers should not.

The `MeanAlgorithm` implementation is thread-safe. The memory complexity is linear with the number of users (one `CachedMean` object per ratings matrix column), when using the mean caching facilities (otherwise the memory usage is constant). The time complexity is constant, but data manager dependent. In case of some direct data manager the time complexity would be linear with the total number of users ($\sim$ read the ratings matrix column and calculate its mean value).

5.4.3 User-User Nearest Neighbours Algorithm

The User-User variant of the (k)-nearest neighbours algorithm (implemented in the `UUNearestNeighboursAlgorithm` class) is the most popular collaborative filtering algorithm and it is also one of the most computationally expensive ones. The prediction for the rating $r_{u1,u2}$ is done in these three separate steps:

1. The vector similarities for the ratings of the user `u1` and all the other users are calculated.

2. The final set of neighbours is created with a special **intersection** operation.

3. The resulting prediction is calculated as a neighbours' weighted mean rating for user *u2*, based on the previous steps.

When a prediction for rating $r_{u1,u2}$ is demanded, most likely the next prediction will be again for the same base user `u1`. This implementation focuses on the cases when predictions for one ratings matrix row are needed consecutively. The user `u1` is declared as a **cached working user** and his similarities to *all* the other users are calculated and cached. The `UUNearestNeighboursAlgorithm` class contains a small cache of recent cached working users (LRU style), so sequent similarities calculation (step 1) is mostly bypassed and the values are retrieved from this local cache instead. The size of such local cache depends on the actual application using this algorithm and corresponds to an average number of online users at the same time requesting their recommendations. The real similarities calculation (when needed) benefits from the fact that the data manager returns rows of the ratings matrix as arrays sorted by user id in ascending order. Having two such ratings vectors, a similarity can be calculated with a linear time complexity. Still, it is the most

time consuming part of the implementation as the total number of users is quite large. This users' ratings similarity calculation is implemented in the following method of the `VectorTools` class:

```
static float getUsersSimilarity(SortedAscIntFixedByteArray ratingsU1, SortedAscIntFixedByteArray ratingsU2, int minCommonVotes);
```

The next step is to select the k nearest neighbours, which will be later used for the actual prediction. All the possible neighbours were acquired earlier, but not all of them shall be used. Because the prediction is desired for the user `u2`, only the neighbours which have rated user `u2` can be used. Also only positive similarities are taken into account, because experiments proved it to yield better results (both accuracy and performance). The intersection operation, which finds the k most similar (nearest) neighbours, which have rated user `u2`, is implemented in the following method of the `VectorTools` class:

```
static FixedIntSortedDescFloatArray intersect(SortedAscIntFixedFloatArray cacheU1, SortedAscIntArray ratersU2, int maxNeighbours);
```

The final prediction is then calculated as a weighted average of the nearest neighbours' ratings of user `u2`. The ratings are weighted by calculated similarities between user `u1` and the selected neighbours.

This algorithm has the following five experimental configurable parameters:

- **minRatings** (mR) – the minimum number of given ratings required from a user to allow predictions for him/her
- **minCommonVotes** (mCV) – the minimum required common non-empty items in two ratings vectors to successfully calculate their similarity
- **minNeighbours** (minN) – the minimum required size of the final neighborhood
- **maxNeighbours** (maxN)[13] – the maximum used size of the final neighborhood (when the neighborhood is bigger, only the top-maxN neighbours are used for a weighted average calculation)
- **cacheSize** (cS) – the size (the number of users) of the local working users cache

The `UUNearestNeighboursAlgorithm` implementation is thread-safe. The memory complexity depends linearly on the total number of users in the system. For C cached active users and their similarities, the memory complexity is $O((C+1) \cdot N)$[14], where N is the total number of users.

The time complexity of a single rating prediction is formed by a time complexity of the similarities calculation, a time complexity of the intersection operation and a time complexity of the final weighted average calculation. The most "expensive" is the initial

[13] For this implementation of the (k)-nearest neighbours algorithm, the k parameter in the algorithm's name actually vary on the real data ($minN \leq k \leq maxN$).

[14] $(C+1) \sim$ the entire cache plus the algorithm's structures

similarities calculation. The similarity of two ratings vectors can be calculated in $O(N)$ or more precisely $O(R/N)$, when assuming sparse and uniform matrix with R ratings and N users. For a prediction of rating $r_{u1,u2}$, a total number of $(N-1)$ similarities have to be calculated (ratings of the user `u1` against the ratings of all the other users). This sums up into the time complexity of the similarities calculation as $O(N^2)$ or $O(R)$. The time complexity of the intersection operation is $O(\log N \cdot R/N)$, because there are R/N scores for user `u2` and for each such score a field in the cached similarities has to be found using binary search. The time complexity of the final weighted average calculation is $O(k)$ as the algorithm uses k nearest neighbours (typically $k \ll N$, so it could be also omitted as $O(1)$). The total time complexity of a single rating prediction is therefore $O(R)$ for users not yet in the cache, and $O(\log N \cdot R/N)$ for already cached user *u1*.

Now, as in every algorithm implementation based on the `AlgorithmAdapter`, the recommendation of top Q users has to do about $(N - R/N)$ predictions to complete the row of the ratings matrix to be able to choose the best users to recommend. This would mean that the time complexity of a single recommendation is $O(N \cdot R + N \cdot \log Q) \sim O(N \cdot R)$[15]. Luckily, the similarities have to be calculated only once (thanks to the similarity cache), so the real time complexity is $O(R \cdot \log N)$.

5.4.4 Item-Item Nearest Neighbours Algorithm

The Item-Item variant of the (k)-nearest neighbours algorithm (implemented in the `IINearestNeighboursAlgorithm` class) is a "mirror" of the User-User variant, so the implementation basics are mostly the same.

Instead of calculating similarities over the ratings vectors (the rows of the ratings matrix), the scores vectors (the columns of the ratings matrix) are used. An important implementation difference is the absence of the similarities cache, as there is no assumption that predictions for one ratings matrix column would be needed consecutively. When predicting the rating $r_{u1,u2}$, the similarities are calculated between the scores vector for user *u2* and the scores vectors for other users. Because of that, there is no reason to have the similarities cached as the recommendation operation needs to predict the complete ratings matrix row anyway.

The scores similarities are calculated each time a prediction is requested. On the other hand, the absence of the cache means that the similarities do not have to be calculated with all the other users as in the User-User variant. In this case, the intersection operation only needs similarities between the scores vector for user *u2* and the scores vectors for users rated by user *u1* (when predicting $r_{u1,u2}$). That is especially useful for data sets with many users and a low mean number of rated users in general[16]. Also without the cache, the Item-Item variant does not have to do any recalculations during the ratings matrix updates.

[15] Q is insignificant ($Q \ll N$).

[16] The LíbímSeTi data set is a great example with about 200,000 users and only 60 given ratings per user in average.

There are two different similarity metrics used: a classic Pearson's correlation coefficient and its adjusted variant. Both of them are very similar and both of them are implemented in the `VectorTools` class:

```
static float getUsersSimilarity(SortedAscIntFixedByteArray scoresU1, SortedAscIntFixedByteArray scoresU2, int minCommonVotes);
```

```
static float getUsersSimilarityAdjusted(DataManager dataManager, SortedAscIntFixedByteArray scoresU1, SortedAscIntFixedByteArray
    scoresU2, int minCommonVotes);
```

In contrary to the User-User configurable parameters stated above[17], the *cacheSize* parameter is not used because there is no cache and one more parameter is added:

- **adjusted** – boolean parameter whether to use adjusted similarity metric or just a classic version of the Pearson's correlation coefficient

The `IINearestNeighboursAlgorithm` implementation is thread-safe. The memory complexity is linear with the total number of users as the similarities have to be calculated and locally stored (even though there are no similarities cached in the process).

The time complexity for a single rating prediction is obtained in very similar way as for the User-User variant. In the first step, only R/N similarities have to be calculated, which means the time complexity $O(R)$ or more precisely $O((R/N)^2)$ for sparse and uniform ratings matrix. The intersection operation in fact just selects top-k neighbours from the calculated similarities, which is done in $O(R/N \cdot \log k) \sim O(R/N)$. The final weighted average calculation is done exactly the same way as in the User-User variant. All this sums up into the total time complexity of a single rating prediction of $O((R/N)^2)$.

When performing the recommendation, again about $(N - R/N)$ predictions have to be calculated to complete the row of the ratings matrix. The real time complexity of a single recommendation is therefore $O(R \cdot R/N)$.

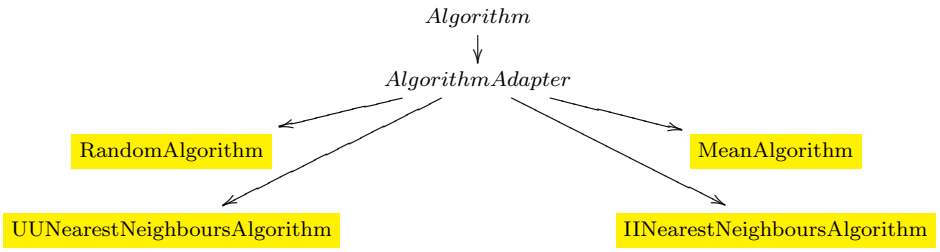


Figure 5.7: `Algorithm` implementations hierarchy

This diagram shows the algorithm classes hierarchy (highlighted classes are final algorithms ready for use, the rest are interfaces or abstract adapter classes).

[17] In case of the Item-Item variant, *minScores* (mS) is used instead of *minRatings* (mR).

Chapter 6

Benchmarks

This chapter describes the benchmarks of all the presented algorithms (implementations) and compares their performance and accuracy on several test data sets.

The purpose of this chapter is to compare implemented collaborative filtering and recommender algorithms described in previous chapters and benchmark them both standalone and as a part of the complete ColFi System solution. All the algorithms shall go through the same set of test cases with the same input conditions (including the test computers' hardware/software configuration). Most of the presented benchmarks are very complex and the test data sets are quite large, so a distributed framework[1] had to be implemented to successfully realize all the tests. The Charles University lab computers together with some other voluntary machines were used to form a distributed environment of about 100 computers[2] to run the presented benchmarks.

In the first part, the automated *laboratory* simulation benchmarks are described, while the second part summarizes some real world experimental results based on the ColFi System deployment in the LíbímSeTi web application environment.

6.1 Simulation Benchmarks

The simulation benchmarks are automated test cases with an explicit design based solely on the data contained in the ratings matrix of the test data set. The result of such simulation benchmark test case shall be some simple value or graph that can be easily interpreted and compared to other similarly obtained results.

This part purposely focuses merely on the simulation benchmarks of the *prediction* abilities of the algorithms as the *recommendation* function is implemented equally for all of them and is based on that prediction. There is a possibility for some algorithm to have a bad absolute prediction results and still be a very good recommendation algorithm, but

[1] The fragments of all benchmarks implementation code are located in the `colfi-benchmarks` package.

[2] Most of the machines used were: PC (x86), Pentium 4 2,6/2,8/3 GHz, 512/1024 MB RAM (dual channel), running the Gentoo Linux distribution.

this fact is out of the scope of this section. Recommendation abilities are benchmarked in a different experimental way in the section 6.2 on the page 58.

6.1.1 Simulation Benchmarks Design

Three different simulation benchmark strategies were chosen to compare the algorithms. Each strategy focuses on some specific feature of the prediction ability:

- **AllButOne** strategy – focuses on the absolute prediction accuracy
- **GivenRandomX** strategy – examines the influence of the amount of initial information given to the algorithm about a single user
- **Production** strategy – tests the overall parameters of the ColFi System solution in the simulated production environment

The first two simulation benchmark strategies are designated to test standalone algorithms and compare mainly their absolute prediction accuracy. Both of them were executed in the distributed environment so only approximate prediction speed measurements were performed as more complex distributed framework would be needed for accurate numbers. The third simulation benchmark strategy (Production strategy) is designated to test the algorithms as a part of the complete ColFi System solution and compare mainly their prediction accuracy and performance/speed over time with concurrent data modifications.

There are 10 algorithms (or algorithm variants) to be tested:

- Random
- Mean (*cache*=off)
- User-User Nearest Neighbours (mR=mCV=5, $minN$=1, $maxN$=1)[3]
- User-User Nearest Neighbours (mR=mCV=5, $minN$=1, $maxN$=10)
- User-User Nearest Neighbours (mR=mCV=5, $minN$=1, $maxN$=50)
- User-User Nearest Neighbours (mR=mCV=10, $minN$=1, $maxN$=1)
- User-User Nearest Neighbours (mR=mCV=10, $minN$=1, $maxN$=10)
- User-User Nearest Neighbours (mR=mCV=10, $minN$=1, $maxN$=50)
- Item-Item Nearest Neighbours (mS=mCV=10, $minN$=1, $maxN$=50)
- Adjusted Item-Item Nearest Neighbours (mS=mCV=10, $minN$=1, $maxN$=50)

[3]The meanings of all the algorithms' parameters are explained in the section 5.4 on the page 36 (mR=min. ratings by user, mS=min. scores for user, mCV=min. common votes for similarity, $minN$=min. required size of the neighborhood, $maxN$=max. used size of the neighborhood).

All the simulation benchmarks have a common base idea. Some part of a test data set gets hidden (the specific part definition is dependent on the selected strategy). The tested algorithm is then asked for a prediction for at least one of the "missing" ratings contained in the hidden part. This is usually done many times with different hidden parts of the test data set, according to the actual strategy. At the end the normalized mean absolute error (NMAE) is calculated over the set of all the obtained predictions in compliance with the equation (6.1):

$$\mathrm{NMAE} = \frac{\frac{1}{c}\sum_{k=1}^{c} |\tilde{p}_{ij}^{k} - r_{ij}|}{r_{max} - r_{min}} \tag{6.1}$$

where c is the total number of ratings predicted in the run, $\tilde{p}_{ij}^{k}$ are those predicted ratings and r_{ij} are the real ratings in the hidden part of the test data set. The denominator represents normalization by the ratings value scale to express errors as percentages of a full scale. The lower the NMAE value gets, the better the algorithm's accuracy is.

As stated above, each simulation benchmarks strategy has its own protocol to define the hidden part of the test data set. These protocols are described in the following sections.

AllButOne Strategy Protocol

The AllButOne strategy protocol is the most commonly used benchmark among many collaborative filtering papers [18, 9]. It is the simplest protocol of the three and it is believed to be the best method for measuring the algorithm's absolute prediction accuracy. The idea behind this protocol is to select exactly one rating and hide it from the tested algorithm. Algorithm is then asked to predict that one hidden rating and the prediction error is the absolute difference between the hidden rating value and the predicted rating value. This is repeated for *every* single rating in the test data set.

It is possible that some algorithms "refuse" to predict certain ratings due to their internal state (mostly due to the absence of information needed for a successful prediction, i.e. too little ratings from active user). These cases are ignored and not included in the final NMAE result[4] and only the total number of such unsuccessful predictions is reported.

The main result of the AllButOne strategy protocol is just one simple NMAE value. The figure 6.1 shows schematic pseudo-code for AllButOne strategy.

```
Initialize the data manager DM with tested data set
Initialize the tested algorithm ALG
For every non-empty rating R from DM do
    Remove R from DM
    Predict R with ALG based on actual DM
    Put R back into DM
EndFor
Calculate resulting NMAE from all successful predictions
```

Figure 6.1: AllButOne strategy pseudo-code

[4]Experiments proved this has no actual impact on the final benchmark result.

GivenRandomX Strategy Protocol

The GivenRandomX strategy protocol is more complex and is intended to examine the algorithm's performance with less active user's data available. This protocol consists of the following steps:

1. Choose one user U with at least 100 ratings given.
2. Hide all the user's U ratings from the algorithm.
3. Randomly select 99 of those hidden ratings and mark them as an ordered group A. The rest of the hidden ratings (given by user U) mark as a group B.
4. For $X : 0 \rightarrow 99$, reveal to the algorithm the first X ratings from the ordered group A and then ask for predictions for all the ratings in the group B. The currently unrevealed ratings from the group A are purposely not used for test predictions together with the group B ratings, as the content of the set of ratings for test predictions should be independent on a given X value (for a comparison to be possible).

These four steps are repeated for *all the users* having at least 100 ratings given. Again, it is possible for any tested algorithm to "refuse" the prediction of any single rating due to its internal state. These cases are ignored and not included in the final NMAE result as in the previous AllButOne strategy protocol.

This strategy reflects the performance of the various algorithms during their startup period, when a user is new to a particular collaborative filtering recommender and show how the algorithms react as user enters more information into the system.

The result of the GivenRandomX strategy protocol is a graph of the NMAE values based on the parameter X (which goes as $X : 0 \rightarrow 99$). The figure 6.2 shows schematic pseudo-code for GivenRandomX strategy.

```
Initialize the data manager DM with tested data set;
Initialize the tested algorithm ALG;
For every user U with at least 100 ratings given do
    RU := Get all the non-empty ratings given by user U from DM;
    Remove complete RU from DM;
    A := Select 99 ratings from RU at random;
    B := RU - A;
    For X := 0 to 99 do
        If X != 0 then
            Put A[X] back into DM;
        EndIf
        For every rating R from B do
            Predict R with ALG based on actual DM;
        EndFor
    EndFor
    Put B back into DM;
EndFor
For X := 0 to 99 do
    Calculate resulting NMAE[X] from all successful predictions under given X;
EndFor
```

Figure 6.2: GivenRandomX strategy pseudo-code

Production Strategy Protocol

The Production strategy protocol is designated to test the algorithm as a part of the complete ColFi System. The entire test data set gets randomly divided into two disjunctive parts[5]: the **snapshot part** and the **simulation part**. Both parts are initially about the same size. The ColFi Server is started with an algorithm being tested, which gets associated with a data manager containing the snapshot part of the test data set. As soon as the system is initialized, the simulation phase begins. At most N_{max} clients (users) concurrently inserts ratings from the simulation part of the test data set into the snapshot part through the ColFi Server interface (this includes additions of new users to the systems as well). Ratings from the simulation part are selected at random. Prior to every single rating insertion on the server side, a prediction of the rating being inserted is computed by the tested algorithm. After every K-sized set of insertions, the NMAE of that set/block is calculated from the obtained predictions.

Although the tested algorithms are issued with the same random sequences of modifications from the clients, the result is not always exactly the same due to a rather chaotic concurrent access in threads. But that is not significant as the number of total modifications/predictions grows very quickly.

The tested algorithms get all the data manager modification events. And it is up to the algorithm implementation whether it discards these events or reacts to them by some sort of internal recalculation (or it may cache some amount of these events and once in a time do the complete recalculation). This benchmark protocol is the most conformable test case to a real world application deployment.

The Production strategy protocol result is a graph of NMAE values based on the index of the K-sized block of insertions. And for the most part, each algorithm's effective implementation speed performance is measured.

6.1.2 Simulation Benchmarks Results

All of the results obtained during the simulation benchmarks are actually in a form of the NMAE (either single value or a graph of some kind). The NMAE equation (6.1), as stated above, is normalized by the ratings value scale. All ratings are internally represented in the scale $\langle 1..127 \rangle$ and therefore all predictions are also in the same scale. It is interesting how much the NMAE results differ if calculated over the internal scale ratings and predictions, or if both ratings and predictions are first linearly converted to the data set's original scale (for example $\langle 1..5 \rangle$). This observation is demonstrated only in the results table of the AllButOne strategy benchmark and the rest of the results are calculated in the ColFi's internal scale[6] $\langle 1..127 \rangle$.

[5]The actual data set division is implemented in the `cz.cuni.mff.colfi.benchmarks.Split` class.

[6]The NMAE values calculated over wider discrete scale should yield more accurate results as the rounding issues will not scatter the predictions that much.

AllButOne Strategy Benchmark Results

All the results of the AllButOne strategy benchmark are summarized in the table 6.1. Each tested algorithm was benchmarked against all four test data sets. Every table cell contains these four fields:

- *two* NMAE values (the bold one is normalized by the ColFi's internal scale $\langle 1..127 \rangle$)
- the total number of unsuccessful predictions during the benchmark run
- the approximate algorithm implementation speed (in predictions per second)

The speed results are omitted for the trivial algorithms (Random and Mean) as their implementations do predict in constant time and the performance numbers gathered in the distributed environment would be completely inaccurate.

The results table reveals few interesting things. First, an expected observation, that bigger neighborhood results in a better accuracy of the User-User algorithm. Further experiments did confirm that more then 50 neighbours would not significantly improve the efficiency (this was also presented in [18]). That is also the main reason why only such Item-Item algorithm variants with reasonable neighborhood sizes of 50 were selected for benchmarking.

The speed results show that the algorithm implementation performance is not that much influenced by the size of the neighborhood. This is because the time consuming similarity calculations are the same for arbitrary final neighborhood size and the specific selection of top-N neighbours is then very fast.

Probably the most important thing observed from this benchmark is that the LíbímSeTi data set yields very low NMAE values for practically all the tested algorithms. This may be caused by the simple fact, that the nature of this data set (and presumably any data set in the dating services area) is just quite suitable for collaborative filtering techniques. And although the Item-Item algorithms did better for most of the data sets (or at least in the adjusted variant), it seems that for the LíbímSeTi data set the User-User algorithm has slightly better results and should probably be preferred, as the Item-Item variants are also generally slower (at least in the recommendation).

A surprising result is a very low NMAE value for the simple Mean algorithm for the LíbímSeTi data set. The difference of 2.41% between the Mean algorithm and the best User-User algorithm variant is very low. And according to the fact that the speed and complexity of those two algorithms differ to a great extent, it is probably up to the deployment environment to choose the "more appropriate" one[7].

[7] See the empirical results in the section 6.2 on the page 58 to understand what other important differences there are between the two mentioned algorithms when performing recommendation.

	MovieLens	Jester	ChceteMě	LibimSeTi
Random	**34.92%** 37.82% 0 –	**32.41%** 32.41% 0 –	**36.98%** 38.16% 0 –	**38.57%** 39.72% 0 –
Mean (*cache*=**off**)	**19.45%** 18.67% 114 –	**21.05%** 21.05% 0 –	**26.21%** 26.03% 0 –	**16.48%** 15.69% 24,785 –
User-User Nearest Neighbours (*mR*=*mCV*=**5**, *minN*=**1**, *maxN*=**1**)	**24.42%** 25.50% 320 *104 p/s*	**22.46%** 22.46% 360 *8 p/s*	**22.58%** 22.91% 3,450 *1,165 p/s*	**19.10%** 19.11% 74,560 *48 p/s*
User-User Nearest Neighbours (*mR*=*mCV*=**5**, *minN*=**1**, *maxN*=**10**)	**18.52%** 18.17% 320 *103 p/s*	**17.28%** 17.28% 360 *8 p/s*	**17.71%** 17.52% 3,450 *1,159 p/s*	**15.13%** 14.54% 74,560 *48 p/s*
User-User Nearest Neighbours (*mR*=*mCV*=**5**, *minN*=**1**, *maxN*=**50**)	**17.73%** 17.16% 320 *98 p/s*	**16.66%** 16.66% 360 *8 p/s*	**17.38%** 17.11% 3,450 *1,115 p/s*	**14.56%** 13.85% 74,560 *48 p/s*
User-User Nearest Neighbours (*mR*=*mCV*=**10**, *minN*=**1**, *maxN*=**1**)	**23.82%** 24.84% 357 *104 p/s*	**22.45%** 22.45% 360 *8 p/s*	**21.69%** 22.00% 5,178 *1,278 p/s*	**17.47%** 17.41% 174,352 *61 p/s*
User-User Nearest Neighbours (*mR*=*mCV*=**10**, *minN*=**1**, *maxN*=**10**)	**18.29%** 17.98% 357 *103 p/s*	**17.28%** 17.28% 360 *8 p/s*	**17.46%** 17.29% 5,178 *1,249 p/s*	**14.18%** 13.56% 174,352 *60 p/s*
User-User Nearest Neighbours (*mR*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	**17.63%** 17.11% 357 *102 p/s*	**16.66%** 16.66% 360 *8 p/s*	**17.25%** 16.99% 5,178 *1,193 p/s*	**14.07%** 13.34% 174,352 *60 p/s*
Item-Item Nearest Neighbours (*mS*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	**17.03%** 16.15% 2,296 *117 p/s*	**16.63%** 16.63% 0 *12 p/s*	**16.71%** 16.22% 481 *1,241 p/s*	**15.03%** 14.51% 476,893 *78 p/s*
Adjusted Item-Item Nearest Neighbours (*mS*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	**16.95%** 16.05% 2,587 *28 p/s*	**16.16%** 16.16% 51 *1 p/s*	**16.09%** 15.57% 825 *311 p/s*	**14.60%** 14.06% 252,623 *35 p/s*

Table 6.1: AllButOne strategy benchmark results

Each algorithm evaluated over all test data sets with AllButOne strategy benchmark. Every table cell contains four numbers:

(1) NMAE of the algorithm prediction calculated in the ⟨1..127⟩ scale
(2) NMAE of the algorithm prediction calculated in the data set's original scale
(3) Number of all skipped rating predictions (due to insufficient data provided to the algorithm)
(4) Approximate algorithm implementation speed ($p/s \sim$ predictions per second)

GivenRandomX Strategy Benchmark Results

The results of the GivenRandomX strategy benchmark are presented as NMAE graphs in the figures 6.3, 6.4, 6.5 and 6.6. Each tested algorithm was benchmarked against all four test data sets. The results for the trivial Random algorithm are not displayed[8] as the values are not so interesting and relatively high which would make the final graphs uneasy to read.

The results confirm that using only one single neighbour for User-User (and presumably also for Item-Item) algorithm is not very efficient and principally even worse than a simple Mean algorithm approach. Other algorithms tend to show better behaviour with dramatically decreasing NMAE values for the first few ratings inserted (for low X values). After the user enters about 30-40 ratings, algorithm's accuracy gets pretty much stabilized ($\sim$ algorithm gets to the point where it has enough information available to "classify" the active user).

Except for the relative comparison of the algorithms prediction accuracy with similar results as from the previous AllButOne strategy benchmark, there are another three interesting facts to notice:

- The graph for the ChceteMě data set suffers from the fact that the data set is quite small and there are only very few users with required 100 or more ratings. That is why the resulting graph is little scattered as there is not statistically enough information present.

- Second, the NMAE values do not converge towards the appropriate results from the AllButOne strategy benchmark for higher X values as one might expect. This is caused by a rather restrictive GivenRandomX protocol requirements which lead to a very different and much smaller group of predictions made during the benchmarking. Further experiments did prove that this is the only reason why the values are slightly different.

- For certain algorithms and certain test data sets (namely the User-User variants with little neighborhood used in the LíbímSeTi data set) the NMAE values increase with higher X values. This unexpected behaviour is caused by too low *minimum common votes* parameter together with overall sparsity of the test data sets. More detailed analysis of this result is presented in appendix A on the page 63.

Overall, the Adjusted Item-Item algorithm seems to outperform all the other algorithm variants for all the data sets except the LíbímSeTi data set, for which the User-User algorithm has generally lower NMAE values. As the LíbímSeTi data set is the main target data set in this thesis, the User-User algorithm is probably the better choice of the two.

[8]The actual results for the Random algorithm are mentioned in the caption texts, next to each graph.

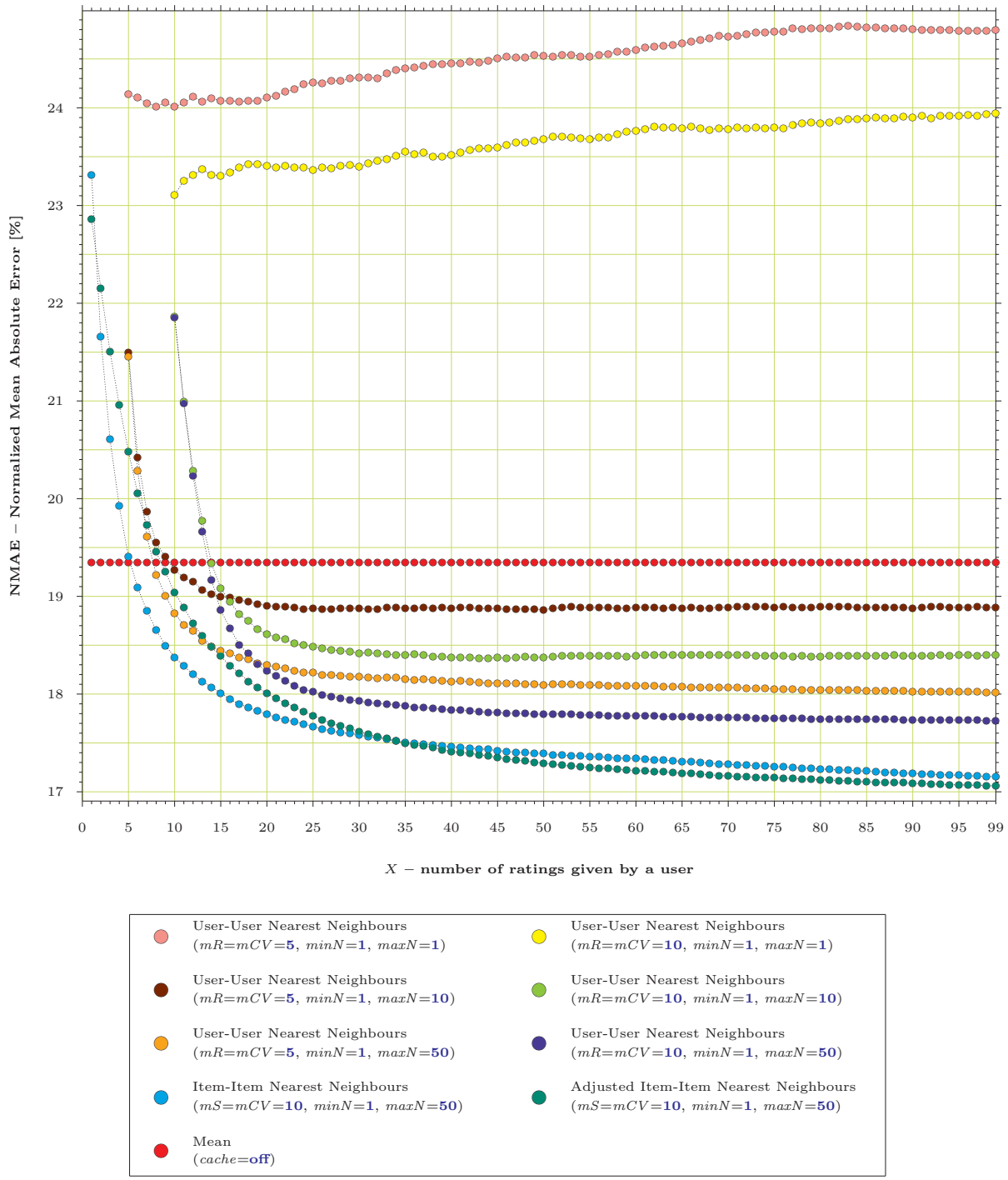


Figure 6.3: GivenRandomX strategy benchmark results - MovieLens

All tested algorithms were benchmarked with the GivenRandomX strategy on the MovieLens data set. The result is a graph of NMAE values based on the number of ratings given by a potential new user. The graph shows the behaviour of the algorithms as they get more information about particular user. Data for the Random algorithm are not displayed as it has relatively high constant NMAE value of 34.46%.

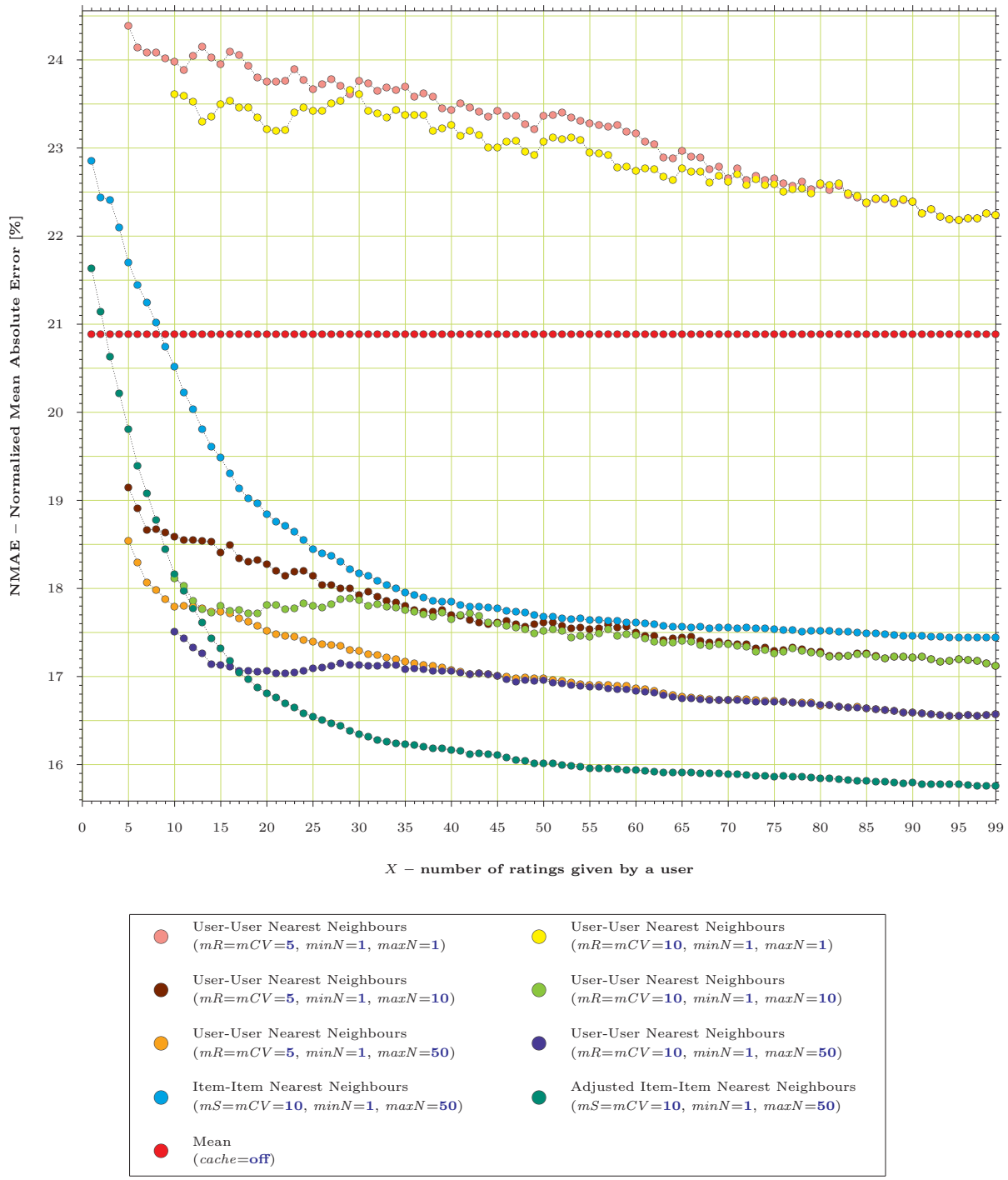


Figure 6.4: GivenRandomX strategy benchmark results - Jester

All tested algorithms were benchmarked with the GivenRandomX strategy on the Jester data set. The result is a graph of NMAE values based on the number of ratings given by a potential new user. The graph shows the behaviour of the algorithms as they get more information about particular user. Data for the Random algorithm are not displayed as it has relatively high constant NMAE value of 32.04%.

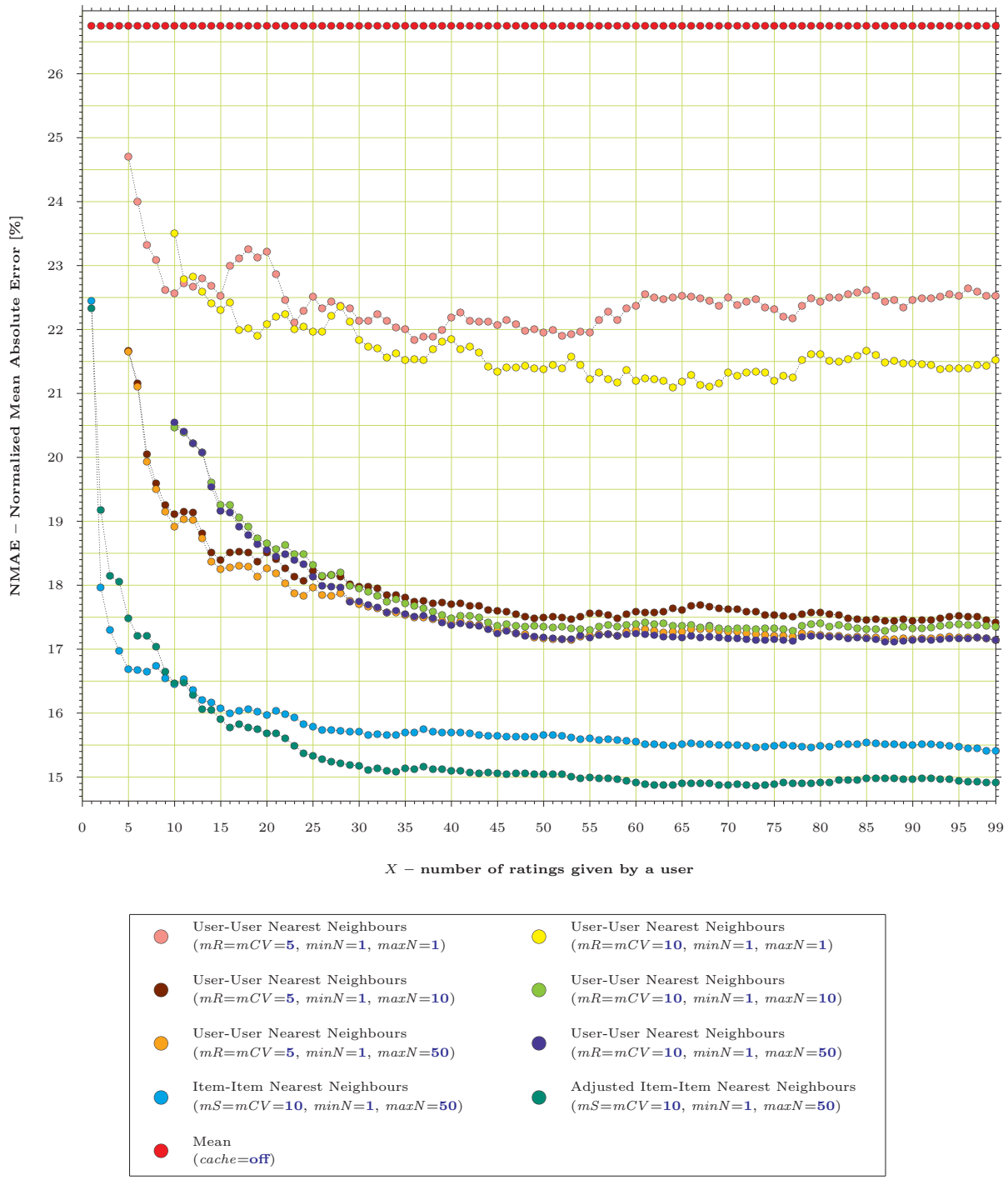


Figure 6.5: GivenRandomX strategy benchmark results - ChceteMě

All tested algorithms were benchmarked with the GivenRandomX strategy on the ChceteMě data set. The result is a graph of NMAE values based on the number of ratings given by a potential new user. The graph shows the behaviour of the algorithms as they get more information about particular user. Data for the Random algorithm are not displayed as it has relatively high constant NMAE value of 37.34%.

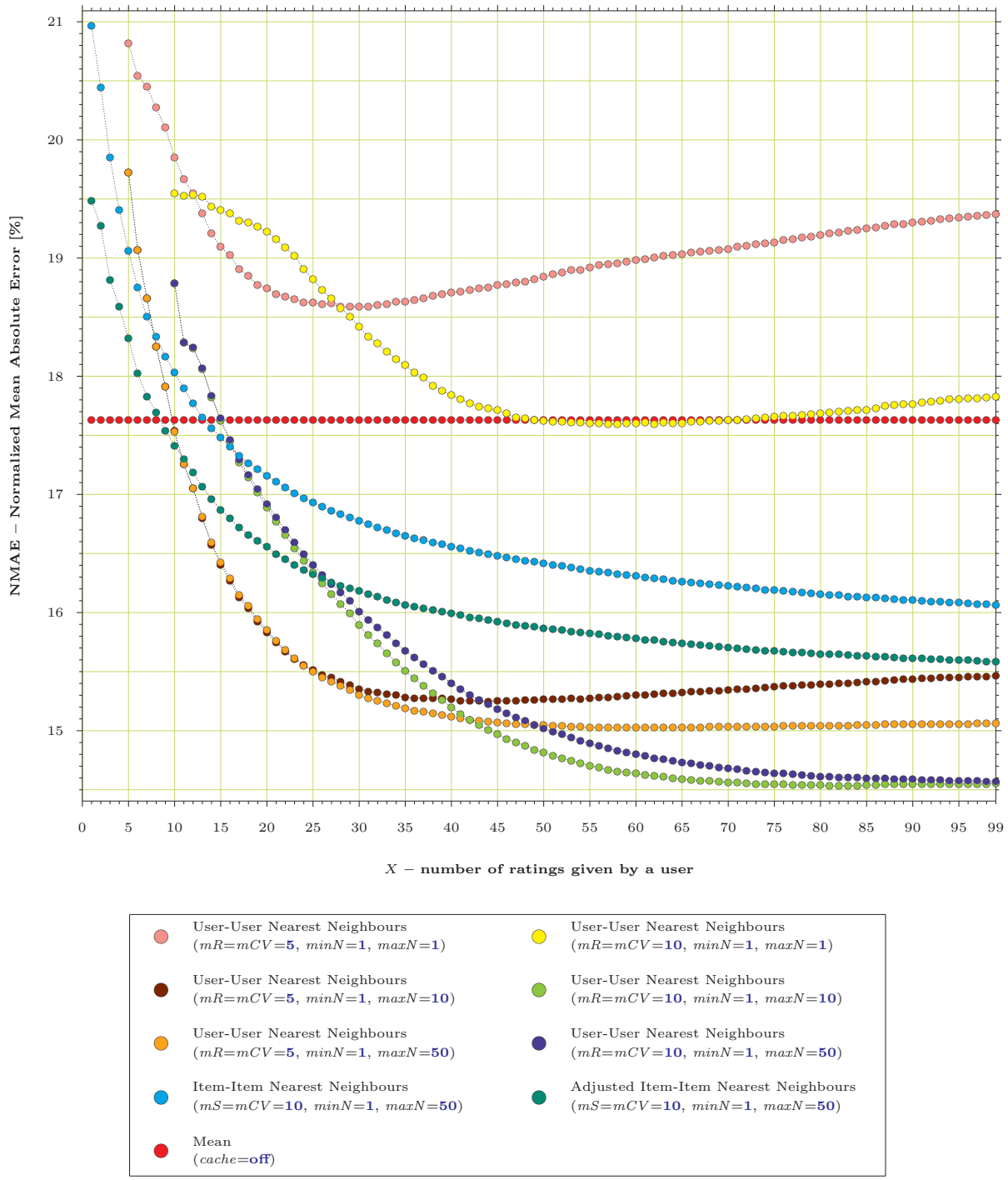


Figure 6.6: GivenRandomX strategy benchmark results - LíbímSeTi

All tested algorithms were benchmarked with the GivenRandomX strategy on the LíbímSeTi data set. The result is a graph of NMAE values based on the number of ratings given by a potential new user. The graph shows the behaviour of the algorithms as they get more information about particular user. Data for the Random algorithm are not displayed as it has relatively high constant NMAE value of 38.27%.

Production Strategy Benchmark Results

The Production strategy benchmark was run on *PC, 2x AMD Opteron™ 250 (2.4GHz, 1MB), 8 GB RAM*, running Red Hat Linux 3.2.3-47. The variable parameters of the protocol were set as follows: $K = 10,000$ and $N_{max} = 100$ ($\sim$ hundred concurrent clients).

The result of the Production strategy benchmark is composed of two parts. First, the NMAE graphs are presented in the figures 6.7, 6.8 and 6.9. And second, the implementation speed performances are summarized in the table 6.2. The ChceteMě data set was not used in this benchmark, because it does not contain enough ratings. The results for the trivial Random algorithm are not displayed in the NMAE graphs[9] as the values are not so interesting and relatively high which would make the final graphs uneasy to read.

As expected, this benchmark did not reveal anything surprisingly new, concerning the NMAE results. Similarly to previous sections, single neighbour variants of the User-User algorithm have significantly higher NMAE values than all the other tested algorithms. And even relative comparison among individual algorithms mainly conform to precedent benchmarks' results. Despite it, there are two important facts for discussion:

- The algorithms' behaviour appears to be slightly unstable with the NMAE values scattered (especially for the LíbímSeTi data set). Yet, the values deviation almost does not exceed 1%, which corresponds to a mean absolute prediction error difference of 1.3 in the ColFi's internal scale. In case of the LíbímSeTi data set and it's original scale $\langle 1..10 \rangle$, that means fluctuation of only about 0.1. These generally small irregularities are caused by a random selection of ratings for insertions together with relatively low K/N_{max} ratio[10].

- Second, it is interesting how similar the NMAE graphs for all the tested algorithms are within each test data set. This observation confirms findings from the previous paragraph, that the graph fluctuations are caused by a random data selection rather than by an algorithm instability.

The table with implementation speed results confirms the estimated time complexities from the section 5.4 on the page 36. The simple Random and Mean algorithms naturally outperform all the other tested algorithms, as they both predict in constant time. All User-User algorithm variants extensively utilize the *local working users cache*[11] as described in the section 5.4.3 on the page 37. This cache is also the reason for lower performance numbers in case of the Jester test data set with high number of valid neighbours, which have to be recalculated every time the cache has to be updated. Item-Item algorithm variants did very good in terms of a single prediction performance, but the recommendation would then use N such predictions, which makes it generally slow. According to these facts and algorithms' accuracy, the User-User algorithm (10-1-50 variant) seems to be the best choice for deployment scenarios similar to LíbímSeTi web application.

[9] The results for the Random algorithm are mentioned in the caption texts, next to each graph.

[10] See the actual implementation in `cz.cuni.mff.colfi.benchmarks.SimulatedClients`.

[11] For this benchmark a cache size of 150 users was used.

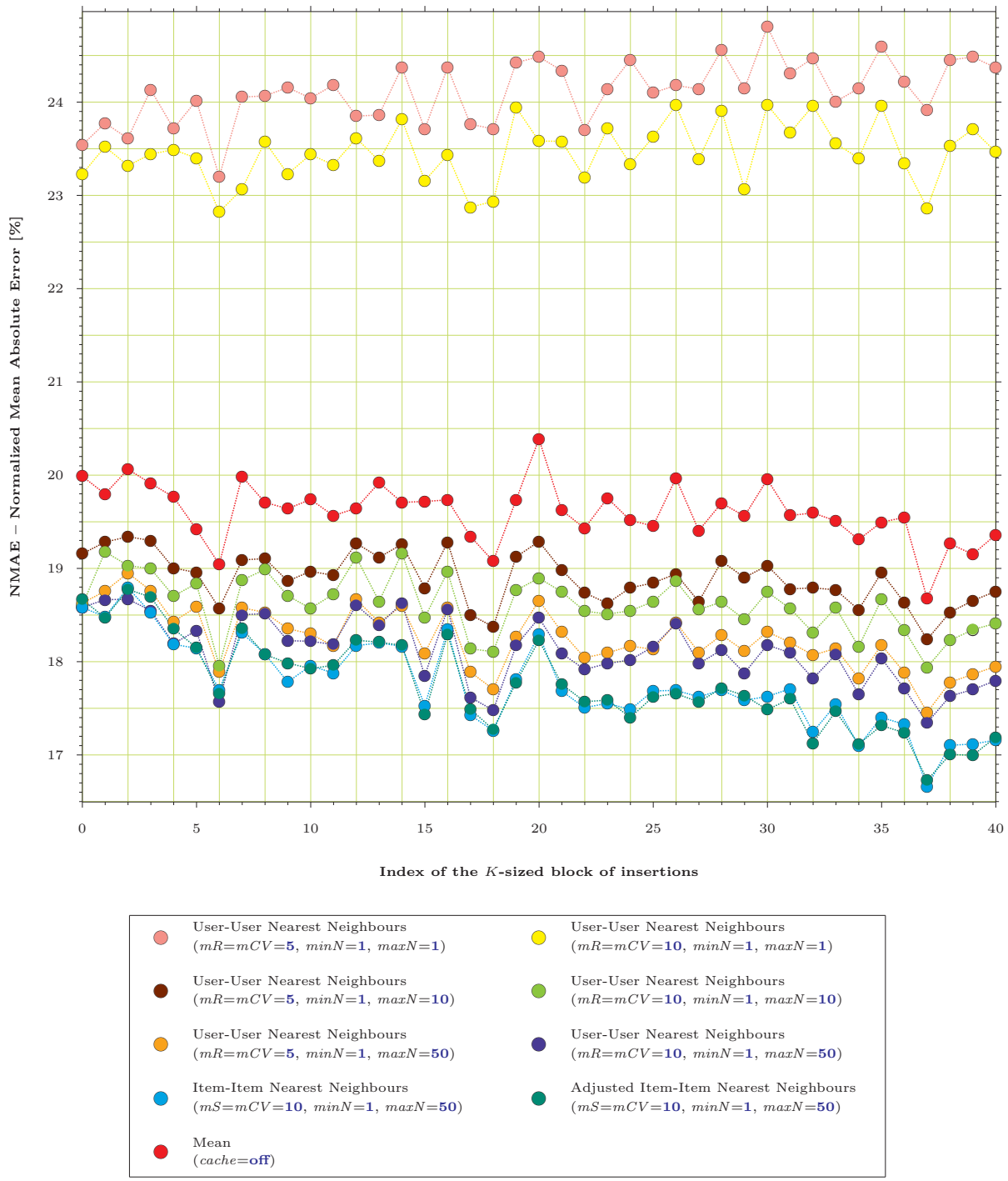


Figure 6.7: Production strategy benchmark results - MovieLens

All tested algorithms were benchmarked with the Production strategy on the MovieLens data set. The result is a graph of NMAE values based on the index of K-sized set/block of insertions. Data for the Random algorithm are not displayed as it has relatively high mean NMAE value of 35.53%.

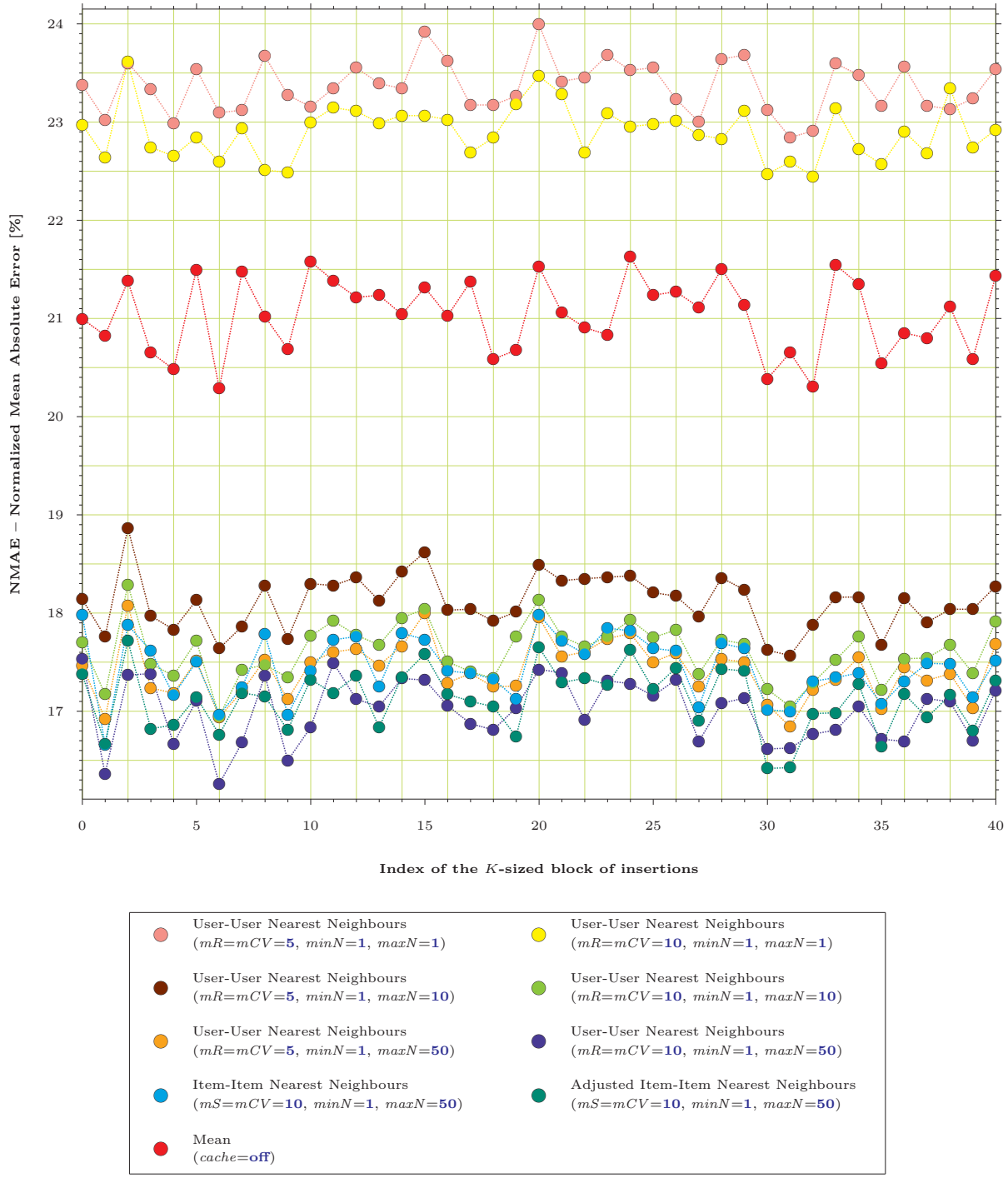


Figure 6.8: Production strategy benchmark results - Jester

All tested algorithms were benchmarked with the Production strategy on the Jester data set. The result is a graph of NMAE values based on the index of K-sized set/block of insertions. Data for the Random algorithm are not displayed as it has relatively high mean NMAE value of 32.44%.

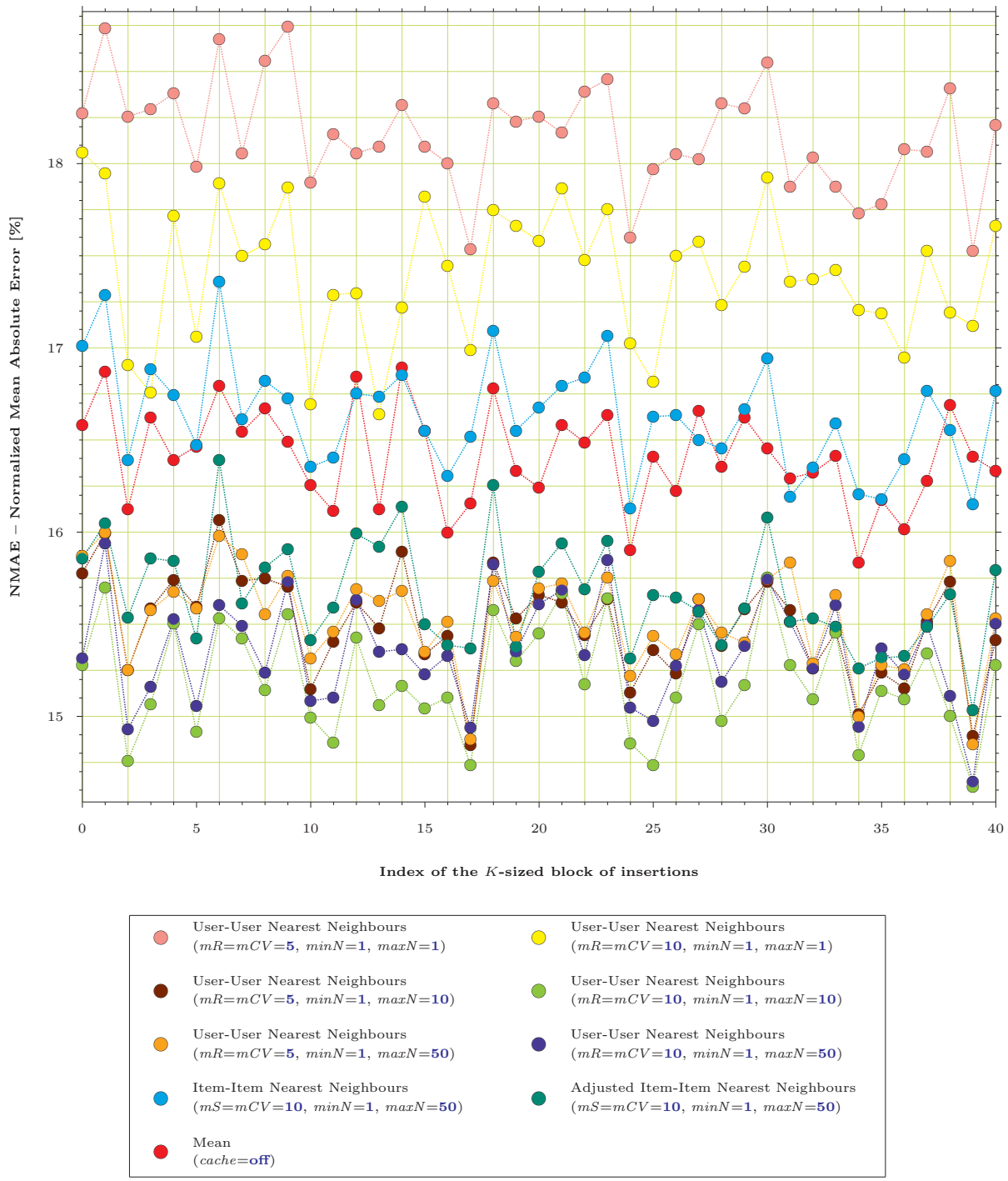


Figure 6.9: Production strategy benchmark results - LíbímSeTi

All tested algorithms were benchmarked with the Production strategy on the LíbímSeTi data set. The result is a graph of NMAE values based on the index of K-sized set/block of insertions. Data for the Random algorithm are not displayed as it has relatively high mean NMAE value of 39.12%.

	MovieLens	Jester	LibimSeTi
Random	18,192	6,784	12,971
Mean (*cache*=**off**)	20,214	6,803	13,175
User-User Nearest Neighbours (*mR*=*mCV*=**5**, *minN*=**1**, *maxN*=**1**)	168	18	103
User-User Nearest Neighbours (*mR*=*mCV*=**5**, *minN*=**1**, *maxN*=**10**)	166	17	98
User-User Nearest Neighbours (*mR*=*mCV*=**5**, *minN*=**1**, *maxN*=**50**)	165	17	87
User-User Nearest Neighbours (*mR*=*mCV*=**10**, *minN*=**1**, *maxN*=**1**)	176	23	113
User-User Nearest Neighbours (*mR*=*mCV*=**10**, *minN*=**1**, *maxN*=**10**)	175	20	113
User-User Nearest Neighbours (*mR*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	158	15	108
Item-Item Nearest Neighbours (*mS*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	324	51	1,264
Adjusted Item-Item Nearest Neighbours (*mS*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	111	3	576

Table 6.2: Production strategy benchmark results

All benchmarked algorithms evaluated over the three main test data sets. Every table cell contains approximate algorithm effective implementation speed in predictions per second (p/s).

6.2 Empirical Benchmarks

The recommendation performance of an algorithm is rather subjective issue to the involved users [21]. Therefore an exact numerical benchmark of the recommendation ability is probably not so accurate[12]. That is the main reason why an empirical approach was chosen instead. As the ColFi System is experimentally deployed in the LíbímSeTi web application environment for testing purposes, it was used to gather some feedback from users.

6.2.1 Empirical Benchmarks Design

Only one empirical benchmark protocol is presented, due to its time complexity. It is called the **LíbímSeTi experiment**. There are 5 algorithms (or algorithm variants) to be tested:

- Random
- Mean (*cache*=off)
- User-User Nearest Neighbours (mR=mCV=10, $minN$=1, $maxN$=50)[13]
- Item-Item Nearest Neighbours (mS=mCV=10, $minN$=1, $maxN$=50)
- Adjusted Item-Item Nearest Neighbours (mS=mCV=10, $minN$=1, $maxN$=50)

Several hundreds of selected users were presented with exactly 150 random user profiles (photos) to rate. After that, each rater was presented with two recommendations from the two distinct randomly selected algorithms being tested. Each recommendation for user `u1` was calculated not from all the users in the data set, but only from the *not rated* users by user `u1`. Every recommendation contains exactly top 10 users. An additional external information about each profile's user gender was used to recommend only the users in the desired gender (according to the chosen filter).

Every user was then repeatedly asked which recommendation is more relevant to him/her - as shown on the example screen shot 6.10. The benchmark results are summarized in the next section.

6.2.2 Empirical Benchmarks Results

Even though the amount of data gathered during this empirical benchmark is not even close to the sizes of test data sets used in the simulation benchmarks in previous sections, the results are very similar. Table 6.3 summarizes the outcomes of individual

[12] Although the *recall-precision* benchmark is sometimes used [32].

[13] The meanings of all the algorithms' parameters are explained in the section 5.4 on the page 36 (mR=min. ratings by user, mS=min. scores for user, mCV=min. common votes for similarity, $minN$=min. required size of the neighborhood, $maxN$=max. used size of the neighborhood).

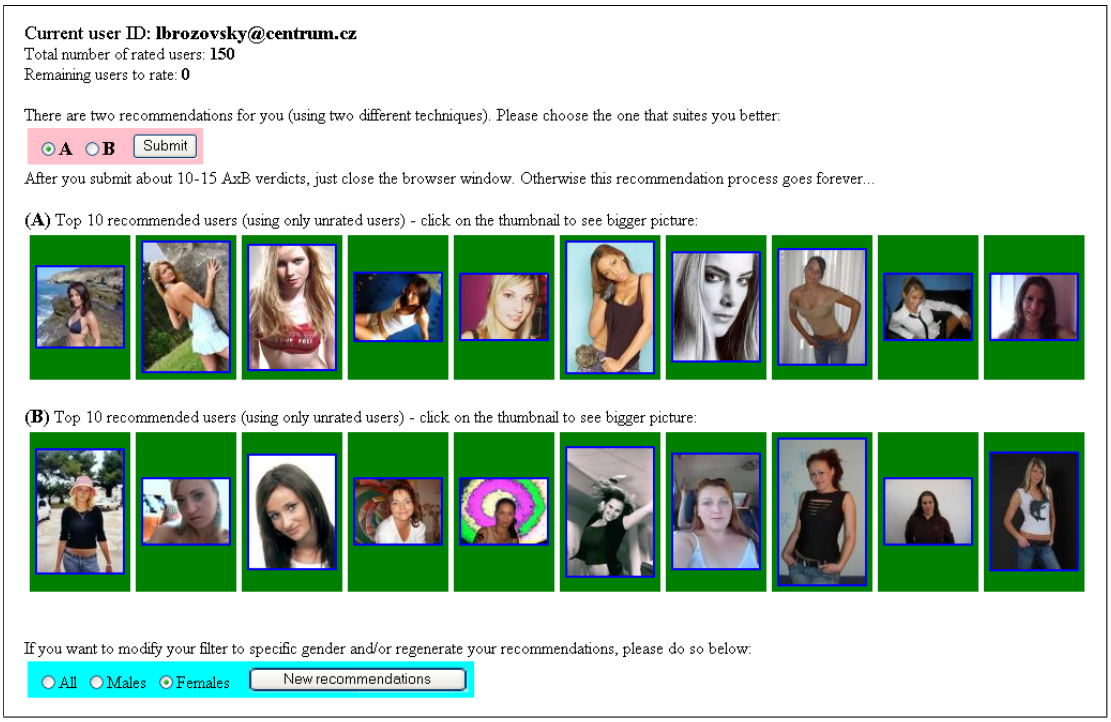


Figure 6.10: LíbímSeTi experiment

An example screen shot of the LíbímSeTi experiment user interface (the recommendation phase).

"recommendation duels" between all tested algorithms. The highlighted cell means that the algorithm A is preferred over the algorithm B by a majority of users.

There are few interesting results observed:

- The Random algorithm naturally "lost" against all the other collaborative filtering algorithms. On the other hand, the numbers in the table for the Random algorithm are quite high, as for example only 3 out of 4 people favour the sophisticated adjusted variant of the Item-Item algorithm over the simple Random approach.
- By numbers, the User-User algorithm "won" against all the other algorithms, although it was close with both Mean and the basic Item-Item algorithm.
- Mainly the high numbers for the Mean algorithm seem to prove that there exists a strong *universal* preference pattern on which most of the users agree.

Nevertheless, the important fact is that the Mean algorithm actually entirely ignores ratings of the active user, which results in the equal recommendations for all users. That is a great limitation for a real life application. According to this observation and the overall benchmark results, the User-User nearest neighbours algorithm variant is the best choice among all tested algorithms.

Algorithm A \ *Algorithm B*	**Random**	**Mean** (*cache*=**off**)	**User-User Nearest Neighbours** (*mR*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	**Item-Item Nearest Neighbours** (*mS*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	**Adjusted Item-Item Nearest Neighbours** (*mS*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)
Random	-	16.87%	12.50%	21.13%	25.88%
Mean (*cache*=**off**)	83.13%	-	35.62%	55.71%	68.67%
User-User Nearest Neighbours (*mR*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	87.50%	64.38%	-	63.53%	78.02%
Item-Item Nearest Neighbours (*mS*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	78.87%	44.29%	36.47%	-	58.06%
Adjusted Item-Item Nearest Neighbours (*mS*=*mCV*=**10**, *minN*=**1**, *maxN*=**50**)	74.12%	31.33%	21.98%	41.94%	-

Table 6.3: Empirical benchmark results

This table shows the outcomes of individual "recommendation duels" between all tested algorithms. Each table cell contains the winning rate of algorithm A over algorithm B. All cases where algorithm A was generally preferred over algorithm B are highlighted ($> 50\%$).

Chapter 7

Conclusion

This concluding chapter summarizes the contributions of the thesis and proposes possible directions for future work.

7.1 Contribution

This thesis was concerned with application of collaborative filtering approach in the domain of online dating services. The main goals and structure of the work were:

- General collaborative filtering research
- Application of collaborative filtering concept to the online dating services environment
- Open-source recommender system implementation
- Benchmarks of the presented solution on the real dating service data sets

All these goals were successfully accomplished. The general collaborative filtering research is summarized in chapters 1 through 3, where the concept of collaborative filtering is described together with several collaborative filtering algorithms. The application of collaborative filtering concept to the online dating services environment is presented as a detailed dating service data sets analysis. Chapters 4 and 5 deal with the implementation of the open-source recommender system[1] (the ColFi System). And finally, the benchmarks of the presented solution on the real dating service data sets were realized in chapter 6.

Interesting results gathered in this thesis proved that collaborative filtering is a viable promising approach that could simplify and improve the matchmaking process in online dating services with a relatively little effort. And even though collaborative filtering algorithms are quite computationally expensive, for their application in dating services environment the matchmaking process does not necessarily have to be real-time, as offline recommendation computations are sufficient.

[1] Open-source recommender system ColFi - http://sourceforge.net/projects/colfi/

7.2 Future Work

The application of collaborative filtering is only the first important step in improving the online dating services. There are several more issues that need to be worked on in the future:

- **Scalability** – presented basic collaborative filtering algorithms are too general and therefore unoptimized and computationally expensive. More advanced algorithms should be tested or developed to answer the problem of growing number of users and huge data sets.
- **Distributed solution** – another answer to the scalability issue could be a distributed recommender system as most of the computations could utilize massive parallelism.
- **Hybrid algorithms** – there is an extensive area of so called *hybrid collaborative filtering algorithms*. These algorithms are combining the collaborative filtering approach together with a simple content-based search.
- **Bilateral matchmaking** – the problem with relationships is that "A likes B" does not imply "B likes A". Therefore each user should be probably presented with recommendations of such users, who are also interested in him/her. This is an important fact, which needs to be researched and maybe some specific algorithms, little beyond collaborative filtering, could be developed.

□

Appendix A

Detailed GivenRandomX Analysis

The following paragraphs discuss the results of the GivenRandomX strategy benchmark presented in the section 6.1.2 on the page 48, mainly the results for the LíbímSeTi test data set observed in the figure 6.6.

At least three of the tested algorithms tend to have increasing NMAE values for higher X values. In other words: algorithm's accuracy decreases with more data inserted by an active user. This unexpected behaviour is caused by too low *minimum common votes* ($\sim$ MCV) algorithm parameter together with overall sparsity of the test data set.

With every inserted rating by an active user, the total number of possible neighbours increases dramatically (mainly because of the low MCV). For each prediction, the User-User variant of the (k)-nearest neighbours algorithm selects the top-k (at most k) most similar neighbours to the active user. The Pearson's Correlation Coefficient vectors similarity function is designed in such a way that it tends to return higher values for couples of vectors with smaller number of common items/votes (small overlaps)[1]. For example with $MCV = 5$, the overlap of five votes is enough to calculate the similarity and there is a great chance that two users do completely agree on their ratings for those five users (the overlap). But in fact, such a neighbour could have completely different preferences for users outside the overlap. On the other hand a "twin" user with almost the same ratings as the active user, but with the overlap, for example, 50 votes, will never achieve extremely high nor perfect similarity.

All these facts lead to a state when the algorithm prefers neighbours with small ratings overlaps and high similarity coefficients over the "actually good neighbours" with larger ratings overlaps and not as high similarity coefficients. That is why the algorithm have worse accuracy results for higher X values, where the number of possible neighbours is very high and this effect becomes evident.

A simple answer to this problem is avoiding extremely low MCV for sparse data sets with low ratings overlaps. Another solution might be weighting the similarity function with the actual overlap size.

[1] This fact may also be observed in the section 2.5.3 on the page 13

Bibliography

[1] Wikipedia on dating service, 2006.

[2] Wikipedia on information overload, 2006.

[3] Wikipedia on information retrieval, 2006.

[4] Wikipedia on matchmaking, 2006.

[5] Andrew Fiore And. Online personals: An overview.

[6] M. Anderson, M. Ball, H. Boley, S. Greene, N. Howse, D. Lemire, and S. McGrath. Racofi: A rule-applying collaborative filtering system, 2003.

[7] P. Baudisch. Joining collaborative and content-based filtering.

[8] Shai Ben-David. A framework for statistical clustering with a constant time approximation algorithms for k-median clustering.

[9] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. pages 43–52.

[10] Christopher N. Carlson. Information overload, retrieval strategies and internet user empowerment, 2003.

[11] Sonny Han Seng Chee, Jiawei Han, and Ke Wang. Rectree: An efficient collaborative filtering method. In DaWaK '01: Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery, pages 141–151, London, UK, 2001. Springer-Verlag.

[12] Sonny Han Seng Chee, Jiawei Han, and Ke Wang. RecTree: An efficient collaborative filtering method. Lecture Notes in Computer Science, 2114:141–??, 2001.

[13] Mark Connor and Jon Herlocker. Clustering items for collaborative filtering.

[14] Dan Cosley, Steve Lawrence, and David M. Pennock. REFEREE: An open framework for practical testing of recommender systems using researchindex. In 28th International Conference on Very Large Databases, VLDB 2002, Hong Kong, August 20–23 2002.

[15] Danyel Fisher, Kris Hildrum, Jason Hong, Mark Newman, Megan Thomas, and Rich Vuduc. SWAMI: a framework for collaborative filtering algorithm developmen and evaluation. In Research and Development in Information Retrieval, pages 366–368, 2000.

[16] Z. Ghahramani and M. Jordan. Learning from incomplete data, 1995.

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

[18] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. Information Retrieval, 4(2):133–151, 2001.

[19] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: an efficient clustering algorithm for large databases. pages 73–84, 1998.

[20] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In SIGIR '99: 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 Press.

[21] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. ACM Trans. Inf. Syst., 22(1):5–53, 2004.

[22] H. V. Jagadish. Linear clustering of objects with multiple attributes. SIGMOD Rec., 19(2):332–342, 1990.

[23] S. Lam and J. Riedl. Shilling recommender systems for fun and profit.

[24] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. IEEE Internet Computing, 7(1):76–80, 2003.

[25] John McCrae, Anton Piatek, and Adam Langley. Collaborative filtering, 2004.

[26] Koji Miyahara and Michael J. Pazzani. Collaborative filtering with the simple bayesian classifier. In Pacific Rim International Conference on Artificial Intelligence, pages 679–689, 2000.

[27] Douglas W. Oard. The state of the art in text filtering. User Modeling and User-Adapted Interaction, 7(3):141–178, 1997.

[28] David 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, UAI 2000, pages 473–480, Stanford, CA, 2000.

[29] David M. Pennock, Eric Horvitz, and C. Lee Giles. Social choice theory and recommender systems: Analysis of the axiomatic foundations of collaborative filtering. In AAAI/IAAI, pages 729–734, 2000.

[30] Daniel Lemi Re. Scale and translation invariant collaborative filtering systems.

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

[32] Paul Resnick and Hal R. Varian. Recommender systems. Commun. ACM, 40(3):56–58, 1997.

[33] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John Reidl. Item-based collaborative filtering recommendation algorithms. In World Wide Web, pages 285–295, 2001.

[34] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John Riedl. Analysis of recommendation algorithms for e-commerce. In ACM Conference on Electronic Commerce, pages 158–167, 2000.

[35] Adam W. Shearer. User response to two algorithms as a test of collaborative filtering. In CHI '01: CHI '01 extended abstracts on Human factors in computing systems, pages 451–452, New York, NY, USA, 2001. ACM Press.

[36] L. Terveen and W. Hill. Beyond recommender systems: Helping people help each other, 2001.

[37] Mark van Setten. Supporting People In Finding Information: Hybrid Recommender Systems and Goal-Based Structuring. PhD thesis, 2005.