Graz University of Technology

Institute for Theoretical Computer Science

Master's Thesis

Comparison of State-of-the-Art Recommender Algorithms

Andreas Töscher

Graz, Austria, January 28th, 2010

Thesis supervisor

Univ. Ass. DI Dr. Robert Albin Legenstein

Technische Universität Graz

Institut für Grundlagen der Informationsverarbeitung

Masterarbeit

Vergleich von modernen Empfehlungsalgorithmen

Andreas Töscher

Graz, Österreich, 28. Jänner 2010

Begutachter

Univ. Ass. DI Dr. Robert Albin Legenstein

Abstract

Recommender Systems are becoming more and more popular these days. They are used in many different areas for recommending books, CDs, movies and other products. The main building block of a recommender system is a collaborative filtering (CF) algorithm, which uses the wisdom of the crowd to predict the taste of the individual. The goal of this thesis is, to compare state-of-the-art collaborative filtering algorithms.

I compare several latent factor models and neighborhood based approaches on the following real world datasets: Jester Joke, MovieLens1M, MovieLens10M and Netflix. For collaborative filtering there exist a lot of popular error measures, which find a widespread use. Therefore I do not restrict my analysis to a single error measure. Instead I use the following four: Mean Absolute Error (MAE), Root Mean Square Error (RMSE), Average Rank (AR) and Area Under Curve (AUC). I empirically investigate the relationships between these error measures. For the latent factor models I investigate the influence of regularization and feature size on accuracy and runtime. Furthermore I investigate the effect of different fillrates of the rating matrix and Gaussian observation noise. The evaluation is done on synthetic data or on the mentioned real world datasets.

My thesis shows that factor models prove to be superior to neighborhood based models on all datasets and with respect to every error measure under investigation. Different fillrates and observation noise do not change the result. Furthermore empirical analysis shows a close relation between the minimas of different error measures.

Keywords. Collaborative Filtering, Recommender System, SVD

Kurzfassung

Empfehlungssysteme werden immer populärer. Sie finden in vielen verschiedenen Bereichen Anwendung, so können sie eingesetzt werden um Bücher, CDs, Filme und andere Produkte zu empfehlen. Der Kernbestandteil eines Empfehlungssystems ist ein collaborative filtering (CF) Algorithmus, welcher aus dem Geschmack von Vielen den Geschack des Einzelnen ableitet. Das Ziel dieser Arbeit ist es, moderne collaborative filtering Algorithmen zu vergleichen.

Hierbei werden verschiedene Latent Factor Modelle und nachbarschaftsbasierte Methoden auf den folgenden Datensätzen verglichen: Jester Joke, MovieLens1M, MovieLens10M und Netflix. Für Collaborative Filtering gibt es einige weitverbreitete Fehlermaße. Aus diesem Grund beschränke ich die Experimente nicht auf ein einziges Fehlermaß, sondern verwende die vier folgenden: Mean Absolute Error (MAE), Root Mean Square Error (RMSE), Average Rank (AR) and Area Under Curve (AUC). Die Beziehungen zwischen diesen Fehlermaßen werden empirisch untersucht. Bei den Latent Factor Modellen wird der Einfluss von Regularisierung und Anzahl der Features auf Genauigkeit und Laufzeit untersucht. Des weiteren werden die Effekte von unterschiedlichen Füllraten der Rating Matrix und Gaußschen Beobachtungsrauschens untersucht. Alle Auswertungen werden auf synthetischen oder den genannten echten Datensätzen durchgeführt.

Diese Arbeit zeigt, dass Latent Factor Modelle bessere Ergebnisse liefern als nachbarschaftsbasierte Methoden, und dies auf allen Datensätzen und bezüglich aller untersuchten Fehlermaße. Unterschiedliche Füllraten und Beobachtungsrauschen ändern dieses Ergebnis nicht. Weiters zeigen empirische Ergebnisse, dass die Minimas verschiedener Fehlermaße eng zusammenhängen.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Contents

List of Figures

Chapter 1

Introduction

Contents

1.1 Overview

Recommender systems use the wisdom of the crowd to make predictions for an individual. So recommender systems can be used in various contexts, where you have lots of users and items. For example they can be used to predict how likely you are going to enjoy a movie or buy a book. A typical webshop has thousands of items and users. Recommender systems can be used to help the user to find items among the overwhelming set of choices.

Recommender systems can learn from different types of information. Typically you group it into explicit and implicit information. Everytime when the user is directly asked to explain his taste to the system, it is called explicit information. The most popular example in this category is the rating. In the case of a 5 star rating scale, the user has to transform his opinion on an item into a 5 star rating. An other popular example for explicit information are rankings, where the user is asked to rank items corresponding to his taste. For implicit information the user is not directly asked. Examples are: The time spent on a page, clicks and purchased products.

In general recommender systems can be categorized into content and collaborative filtering systems.

1.2 Content Filtering

Content filtering uses additional item information for the recommendations. Based on the users historical information on liked and disliked items and item-item similarities, a content filtering algorithm calculates recommendations. In the case of content filtering item-item similarities are based on item meta data. So the quality of a content filtering system strongly depends on the quality of the item meta information. Typically there are two ways of generating item meta data. The first option is the handcrafted generation, which can be very time consuming and error-prone. Typical examples therefore are coarse categorizations of items like genres for movies. The second option for generating item meta information is to automatically generate it. There exist a lot of examples where this has been done for text. A classical example for automatic feature extraction is described in [20], where content filtering is applied to newsgroup messages. For pictures, video and audio data the problem of automatically extracting content information is much harder.

1.3 Collaborative Filtering

Pure collaborative filtering (CF) does not use user and item meta information. The underlying assumption of CF is, that users who agreed on something in the past tend to agree on it in the future. So CF searches for users with similar taste and uses this information to generate predictions.

1.4 Goal of the Thesis

Over the last years lots of collaborative filtering algorithms were published. Most of the papers focus on a specific algorithm on a specific dataset and report results for one specific error measure. This resulted in a variety of different algorithms, datasets and error measures. All of this makes it hard to compare new results to older ones. It is often very hard to judge if an improved result on a benchmark dataset results from algorithmic improvements and prevails for other datasets, or from modelling dataset specific properties. There is also the big question on which error measure to use for comparison since all of the commonly used have their justification and real world relevance.

In the thesis I focus on pure rating based collaborative filtering algorithms, because most published results are stemming from this area. I compare classic neighborhood based methods with matrix factorizations. For comparing the algorithms I use accuracy measures and ranking error measures, and I analyze which are better suited for optimizing and analyzing collaborative filtering algorithms. Then, I analyze the accuracy of the algorithms on the MovieLens1M, MovieLens10M, Jester Joke and Netflix dataset. Furthermore, it will be shown how to generate synthetic data for conducting experiments which would not be possible otherwise. I also take a close look at the behavior of the algorithms for very sparse and non sparse data, as well as at the influence of high observation noise.

The main finding of my thesis is that factor models proved to perform well on all dataset and error measure combinations. Regularization is shown to be important for factor models, which is not very surprising for imbalanced datasets. Different fillrates of the rating matrix and different levels of observation noise do not change a performance based ranking of the algorithms. A very interesting result is that the best algorithms are mostly the same for all error measures. Furthermore, I empirically found that the minima regarding different error measures are closely related.

1.5 Mathematical Notation

Within this work I use datasets with jokes and movies, we will call them allways just items. The set of users is denoted by U and the set of items with I. The set of items rated by user u is I_u, while the set of users which rated item i is called U_i. The rating of user u on item i is written as r_{ui}. One can think of the ratings as being stored in a sparse matrix $\mathbf{R} = [r_{ui}]$ of the size $|U| \times |I|$. The set of user-item tuples with an existing historical rating r_{ui} is called $\mathcal{L} = \{(u,i)|u \in U, i \in I, \text{user u has rated item i}\}$. $\mathcal{L}$ is split into two disjoint sets $\mathcal{T} \cap \mathcal{P} = \emptyset$, the train set $\mathcal{T}$ and the probe set $\mathcal{P}$, so that

$\mathcal{T} \cup \mathcal{P} = \mathcal{L}$. The way how to split away the probe set is reported for every dataset in Chapter 2. Predicted ratings or approximations are denoted as $\hat{r}_{ui}$. All sorts of meta parameters are written as small Greek letters (α, β, γ, ...). Vectors and matrices will be written in bold face. Additional definitions will be made on demand throughout this work.

1.6 Netflix Prize

Netflix Inc. is a leading online movie rental company, which started a world wide competition to improve their own Cinematch algorithm by 10%. The competition started in October 2006 and ended in September 2009. Michael Jahrer and I participated in the competition and achieved to be part of the winning team for the Progress Prize 2008 and the final Grand Prize in 2009. Throughout my thesis I will have remarks and notes on topics where we gained experience from the Netflix Prize.

Chapter 2

Datasets

Contents

2.1 Overview

Data for collaborative filtering is typically collected from explicit or implicit user feedback. Explicit feedback means that the user is asked to explain his opinion to the system. Examples for explicit feedback data are ratings and orderings. Implicit feedback data is collected out of user actions. Examples are clicks, time of viewing or purchase information. The statistical properties of the different sources of information, can be very different. Purchase information is typically a high quality information, whereas click information has low quality. But not only the noisy level is different between the datasets. The user interface design plays a major rule. Sometimes the ratings are missing at random but mostly not. Rating scales can be continuous or discrete, and could have different ranges. Users can be asked to rate an initial fixed set of items. So the datasets can be very different. Obviously this influences the performance of the algorithms, and a good comparison should use as different datasets as possible.

The comparison of the recommender algorithms is done on 4 real world datasets and a synthetic dataset. The 4 real world datasets have different numbers of ratings and different statistical properties. All of these datasets stem from explicit feedback, due to the lack of freely available implicit feedback datasets. Not every algorithmic property of interest can be tested using real world datasets so we also use a adjustable synthetic dataset for some experiments.

2.2 MovieLens

MovieLens[*] is a free online movie recommender system from the GroupLens research group at the University of Minnesota. Users can rate known movies, and based on this information the system recommends new movies to the users. The GroupLens research group provides 3 datasets with 100 thousand, 1 million and 10 million of ratings. We use the 1 million and 10 million datasets. Both datasets use a 5 star rating scale ranging from 1 to 5 stars.

2.2.1 MovieLens1M

The MovieLens1M dataset consists of exactly 1,000,209 ratings of 6,041 users on 3,707 items. The fillrate of the rating matrix is 4.47%. The probe set is a fixed 10% random subset. In Figure 2.1 we can see a overall rating histogram, in Figure 2.2 a histogram of the number of ratings per item and in Figure 2.3 a histogram of ratings per user. These ratings per user and ratings per item histograms are clearly distributed exponentially. This means that few users voted a lot, while most users have given just a few ratings. Also, there are some very popular items which are heavily rated, while the most have few ratings. This imbalancedness of the data is the reason why proper regularization is important. We will discuss this in Section 6.2.

2.2.2 MovieLens10M

The MovieLens10M is the latest and biggest dataset from the GroupLens research group. It has 10,000,054 ratings from 71,567 users on 10,681 items. The fillrate of the rating matrix is 1.31%. The probe set is a fixed 10% random subset. Figure 2.4 shows

[*]`http://movielens.umn.edu`

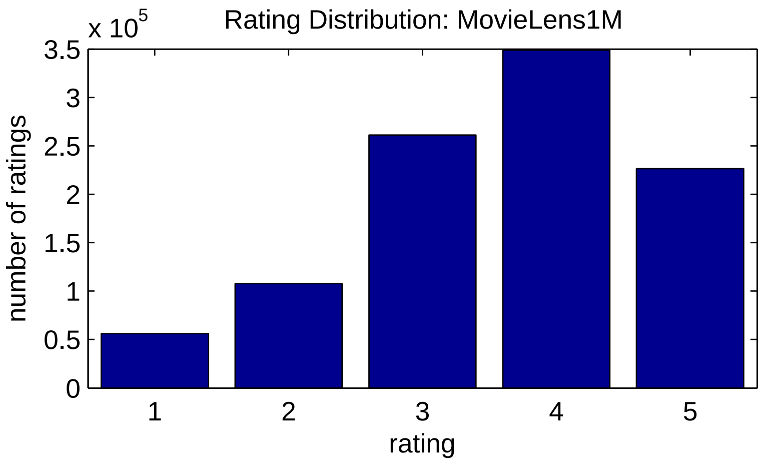

Figure 2.1: MovieLens1M: The rating distribution is clearly shifted towards 3 to 5 star ratings. The 4 star rating is the most prominent.

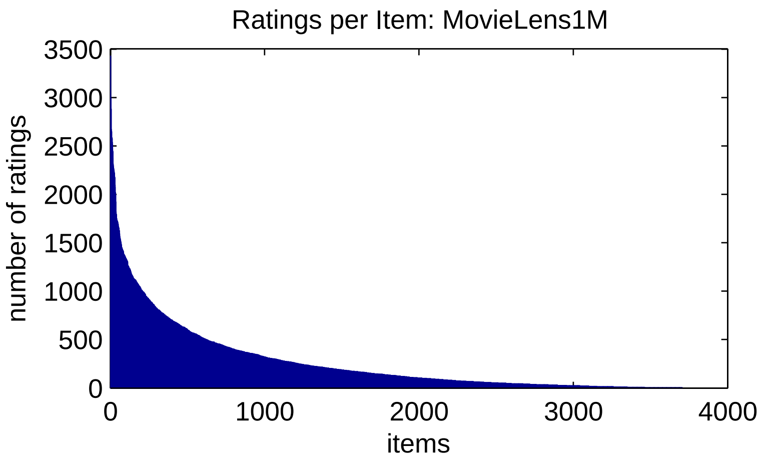

Figure 2.2: MovieLens1M: The number of ratings per items varies over a large scale. Some items got heavily rated, while others are rated rarely.

the rating histogram. The histogram with ratings per item can be seen in Figure 2.5 and the rating per user histogram in Figure 2.6.

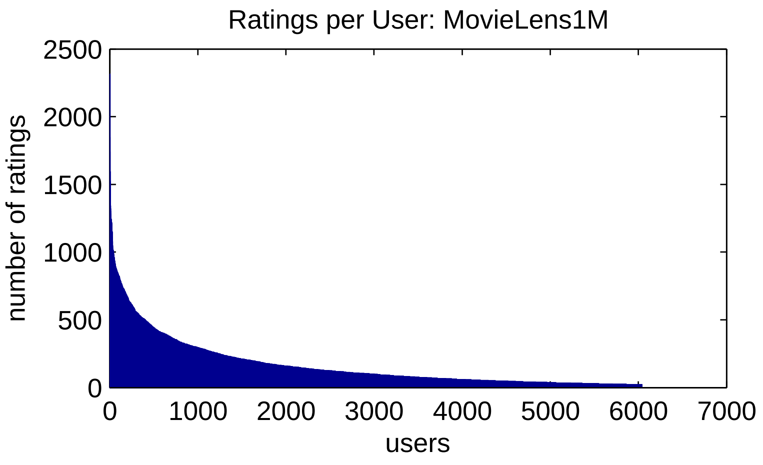

Figure 2.3: MovieLens1M: The number of ratings per user varies as much as the ratings per item, but on a different scale.

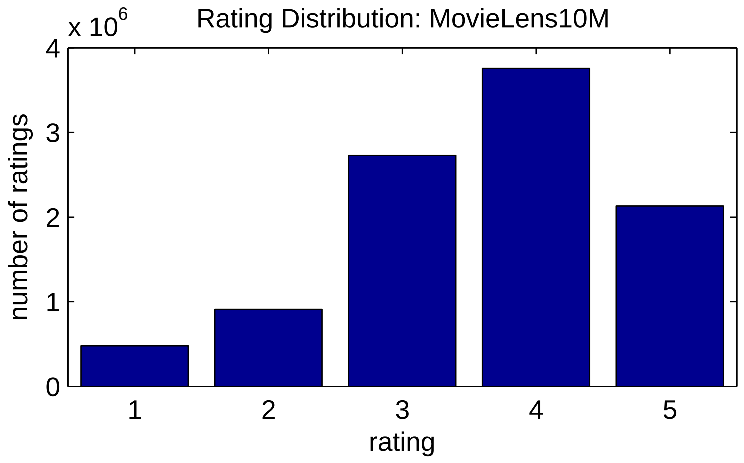

Figure 2.4: MovieLens10M: The rating histogram of the MovieLens10M dataset looks similar to the smaller MovieLens1M. The rating distribution is clearly shifted towards three to five star ratings. The four is the most common rating.

2.3 Jester

Jester is an online joke recommender site[†]. The users can rate jokes on a continuous rating scale from -10 to 10. Based on the users previous ratings a recommender algorithm suggests new jokes to the user. Originally Jester used the Eigentaste algorithm

[†]`http://shadow.ieor.berkeley.edu/humor/`

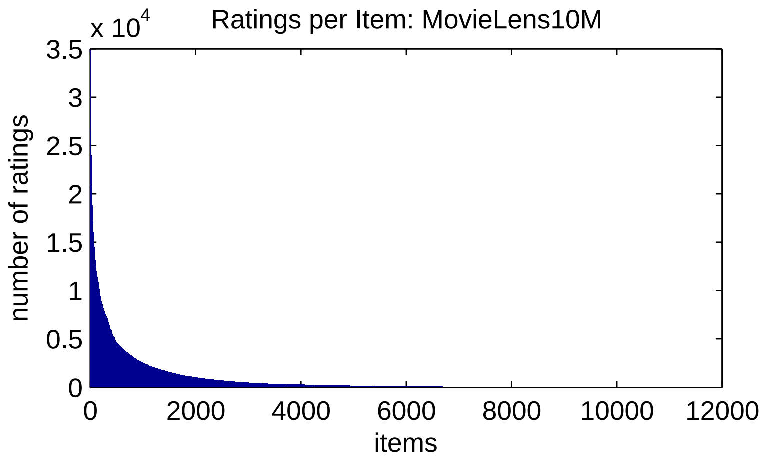

Figure 2.5: MovieLens10M: The number of ratings per item ranges from a few hundred to over 20,000.

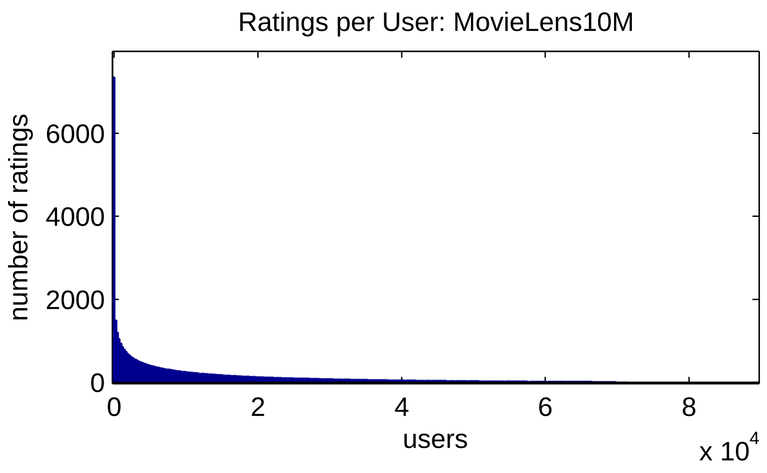

Figure 2.6: MovieLens10M: This histogram shows nicely that the actual number of ratings per user differs widely. A single user rated over 7,300 movies, while the average user rated 140.

as described in [12]. Currently Jester uses the improved Eigentaste 5.0 algorithm [22].

Based on the collected rating data Jester provides a dataset with 4,116,866 ratings from 73,421 users on 100 jokes. Compared to other collaborative filtering datasets the fillrate of this dataset is much higher. In the Jester dataset 56.07% of the elements of the rating matrix are known. In Figure 2.7 you can see a rating histogram. The probe set is a fixed 10% random subset.

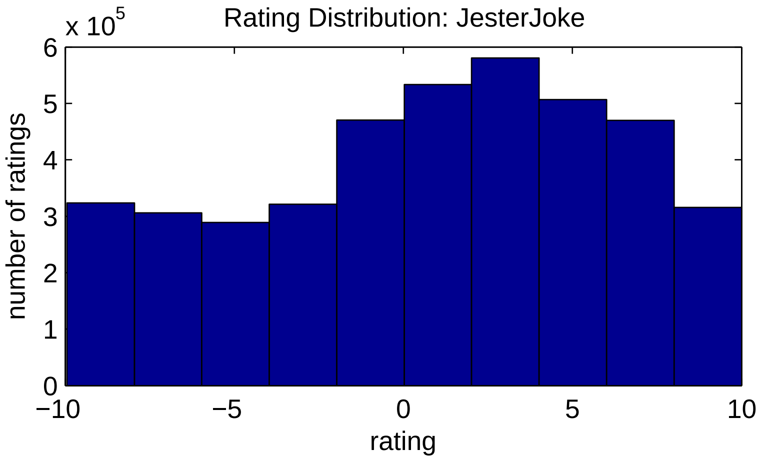

Figure 2.7: Jester: The continuous rating scale ranges from -10 to 10. The rating distribution is more flat as in the MovieLens or Netflix dataset, with a slight overweight of positive ratings.

Figure 2.8 contains the histogram of ratings per item and Figure 2.9 the ratings per user. The Jester website asks every new user to rate the same amount of initial jokes, and starts afterwards to recommend jokes to the users. This behavior is clearly visible in the histograms.

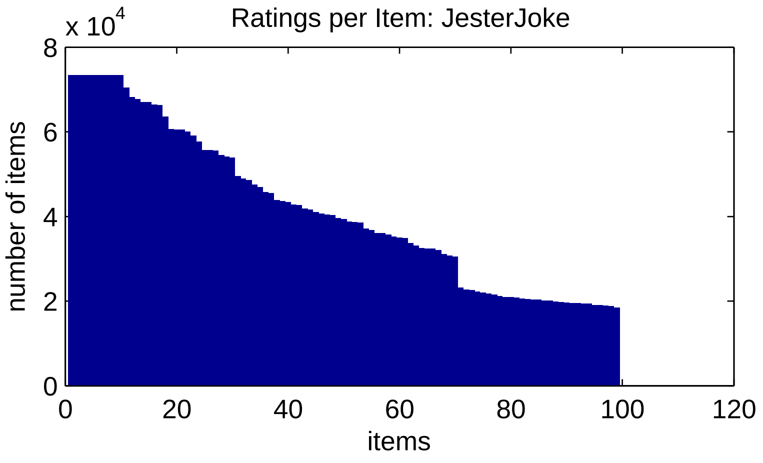

Figure 2.8: Jester: The first few jokes got rated by nearly every user. This property stems from the fact that a new user has to rate this fixed set of jokes first, before the recommender system starts to recommend jokes.

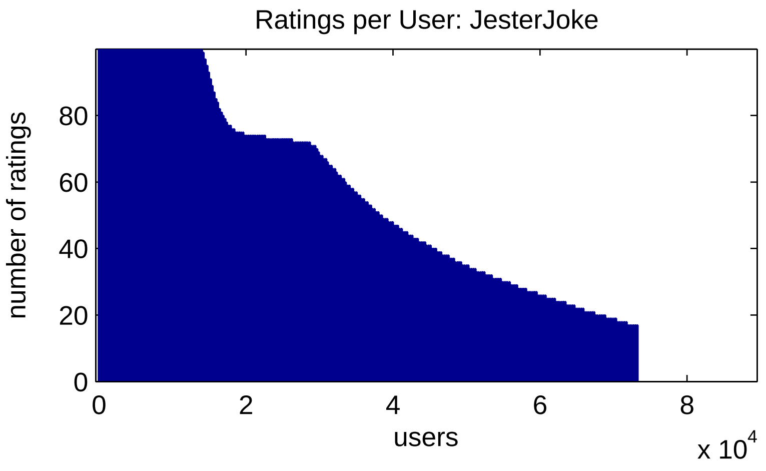

Figure 2.9: Jester: Some users rated every joke, while there are also users who only rated the initial jokes.

2.4 Netflix

Netflix is an online movie rental company. Users can browse and order movies online. A key part of the Netflix service is a recommender system, called Cinematch. In October 2006 Netflix launched a competition[‡] with a prize money of $1,000,000, with the goal to improve the Cinematch algorithm by 10% (measured in terms of the RMSE). For this competition Netflix released a dataset with 100,480,507 ratings of 480,189 users on 17,770 items. Thus the dataset has a fillrate of about 1.17%. The probe set $\mathcal{P}$ used for reporting performance values in this thesis is exactly the same as defined by Netflix for the competition. The rating histogram is shown in Figure 2.10, while the histogram with ratings per item can be seen in Figure 2.11 and ratings per user in Figure 2.12.

2.5 Synthetic Dataset

Experiments on real world datasets are very important, but it is not possible to find real world datasets that cover every property of interest. So we are using a synthetically generated dataset for some later experiments.

For collaborative filtering it is important that the ratings between the users and the items are correlated. For this reason we construct the rating matrix $\mathbf{R}$ by summing up

[‡]`http://www.netflixprize.com`

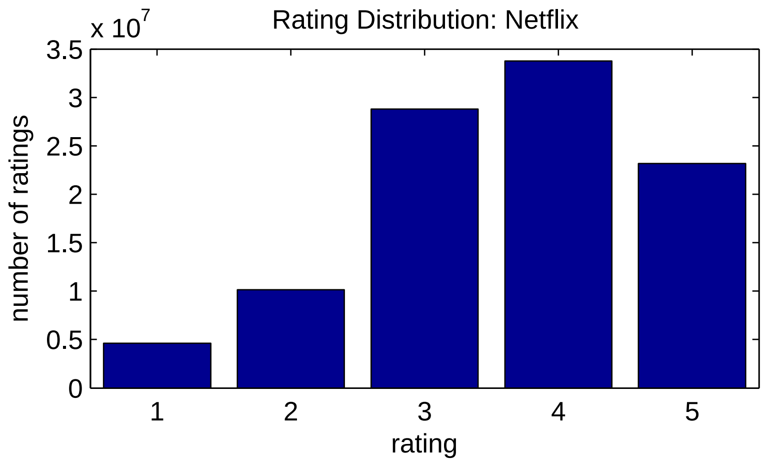

Figure 2.10: The average rating in the Netflix dataset is 3.6, and the most often used rating is the four. The histogram itself is clearly shifted towards the higher ratings, as for the MovieLens datasets.

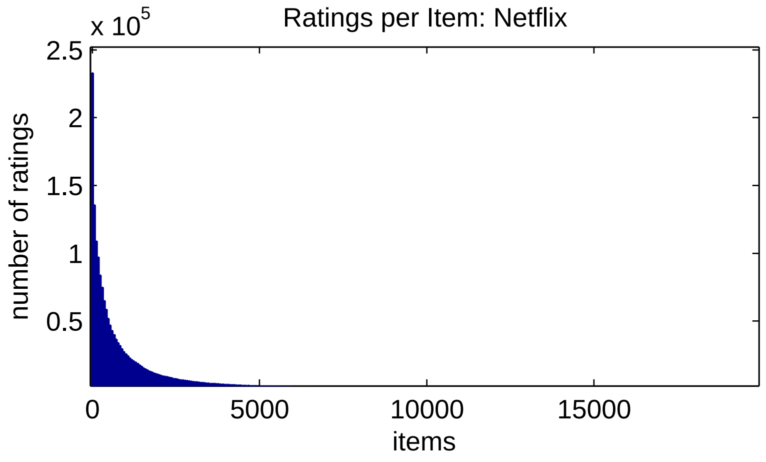

Figure 2.11: Netflix: The ratings per item range from under 10 to over 200,000. So there are very popular and very unpopular items in the dataset. The average item got 5,654 ratings.

the user rating matrix $\mathbf{R}^{\mathbf{user}}$ and item rating matrix $\mathbf{R}^{\mathbf{item}}$ and the noise matrix $\mathbf{\Gamma}$

$$\mathbf{R} = \mathbf{R}^{\mathbf{user}} + \mathbf{R}^{\mathbf{item}} + \mathbf{\Gamma}. \tag{2.1}$$

The user rating matrix $\mathbf{R}^{\mathbf{user}}$ stems from a multivariate Gaussian distribution, with non zero correlations between the users. The same applies to the item rating matrix

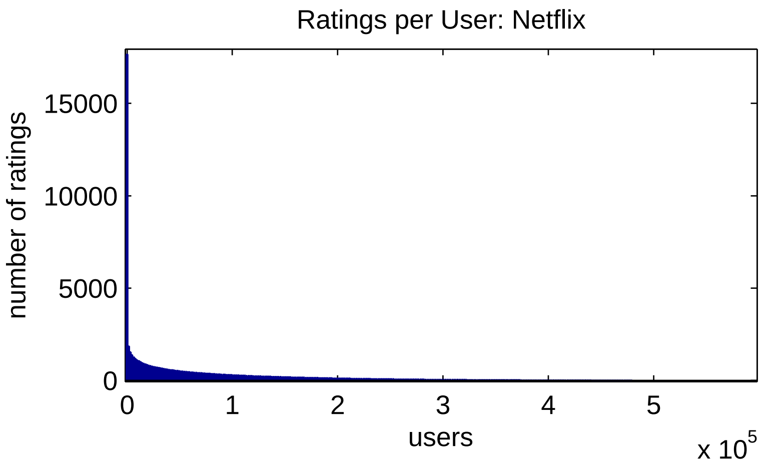

Figure 2.12: Netflix: The ratings per user distribution looks similar to the MovieLens datasets. The ratings per user range from 1 to over 17,000. The average user has rated 209 items.

$\mathbf{R}^{\mathbf{item}}$, which also has non zero correlations between the items.

The noise matrix $\mathbf{\Gamma} = [\gamma_{ij}]$ in Equation 2.1 simulates the rating noise, which is the variability in the user ratings. Within this work we restrict our analysis to Gaussian noise $\gamma_{ij} \sim \mathcal{N}(0; \sigma)$. By using $\sigma = 0$ a user gives always the same rating for an item. Values for σ greater than 0 account for the fact that users do not give the same ratings if they are asked to rerate already rated items. The user rating matrix $\mathbf{R}^{\mathbf{user}}$ has the size $|U| \times |I|$ and consists of $|I|$ column vectors $\mathbf{r}_{\mathbf{i}}^{\mathbf{user}}$ of the dimension $|U| \times 1$. The vectors $\mathbf{r}_{\mathbf{i}}^{\mathbf{user}}$ are drawn i.i.d. from

$$\mathbf{r}_{\mathbf{i}}^{\mathbf{user}} \sim \mathcal{N}(\mu^{user}\mathbf{1}; \mathbf{\Sigma}^{\mathbf{user}}) \tag{2.2}$$

a multivariate Gaussian distribuation with the covariance matrix $\mathbf{\Sigma}^{\mathbf{user}}$, which defines the correlations between the users. The item correlation matrix $\mathbf{R}^{\mathbf{item}}$ consists of $|U|$ row vectors $\mathbf{r}_{\mathbf{u}}^{\mathbf{item}}$ with the dimension $1 \times |I|$ which are drawn i.i.d. from

$$\mathbf{r}_{\mathbf{u}}^{\mathbf{item}} \sim \mathcal{N}(\mu^{item}\mathbf{1}; \mathbf{\Sigma}^{\mathbf{item}}) \tag{2.3}$$

a multivariate Gaussian distribution.

The problem of generating a meaningful rating matrix $\mathbf{R}$ can therefore be reduced to the problem of generating valid user and item covariance matrices $\mathbf{\Sigma}^{\mathbf{user}}$ and $\mathbf{\Sigma}^{\mathbf{item}}$.

Hirschberger et al. [17] describes a method for generating valid covariance matrices by specifying diagonal mean and variance and the offdiagonal mean of $\mathbf{\Sigma}$.

The idea is to construct $\mathbf{\Sigma}$ as a product of the matrix $\mathbf{F}$ with itself:

$$\mathbf{\Sigma}^{\mathbf{user}} = \mathbf{F_u}\mathbf{F_u}^T \tag{2.4}$$

$$\mathbf{\Sigma}^{\mathbf{item}} = \mathbf{F_i}\mathbf{F_i}^T \tag{2.5}$$

Thus the constructed covariance matrix $\mathbf{\Sigma}$ is always positive semidefinite. In order to specify the diagonal mean e, the diagonal variance v and the offdiagonal mean $\bar{e}$ the matrix $\mathbf{F}$ has to be generated as follows:

$$m = round\left(\frac{\bar{e}^2 - e^2}{v}\right) \tag{2.6}$$

$$\hat{e} = \sqrt{\frac{e}{m}} \tag{2.7}$$

$$\hat{v} = -\hat{e}^2 + \sqrt{\hat{e}^4 + \frac{v}{m}} \tag{2.8}$$

$$f_{ij} = \hat{e} + \sqrt{\hat{v}}q_{ij} \tag{2.9}$$

$$q_{ij} \sim N(0;1), i = 1,..,n, j = 1,...,m \tag{2.10}$$

After generating the synthetic rating matrix $\mathbf{R}$ as described, the matrix is fully filled. For CF the rating matrix is typically very sparse, so we have to draw a random subset from the full matrix in order to get a specified fillrate. Every item will have roughly the same number of ratings, while the same is true for the users. Therefore, the user and item rating distributions will be flat, hence differing from the exponential distributions of Netflix and MovieLens. It would be very interesting to introduce parametrized distributions and to investigate the implications for the needed regularization, but this will be left open to future research.

Throughout this work we will refer to the following three predefined synthetic datasets.

2.5.1 Configuration 1

The configuration 1 uses 5,000 users and 1,000 items. The fillrate of the rating matrix is 4% and the rating mean is 3. The rating noise is $\sigma = 0.1$. A random subset of 10% is used as the probe set. The rating histogram is shown in Figure 2.13. The histogram with ratings per item can be seen in Figure 2.14 and ratings per user in Figure 2.15.

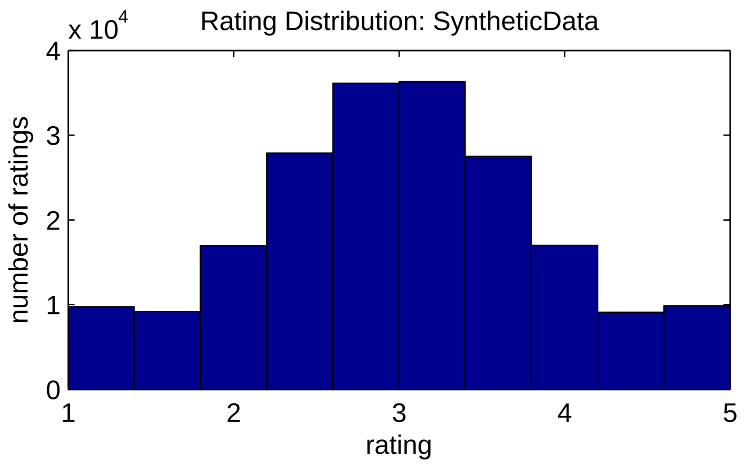

Figure 2.13: Synthetic dataset configuration 1: It can be clearly seen that the global rating mean is at 3.

2.5.2 Configuration 2

The second configuration uses the same number of users and items, as the first configuration. The fillrate with 4% and the mean rating with 3 is also the same. The difference to configuration 1 is the much higher and more realistic rating noise with $\sigma = 0.7$. For generating this dataset the ratings got also rounded to discrete ratings, which introduces additional noise. Figure 2.16 shows the rating histogram.

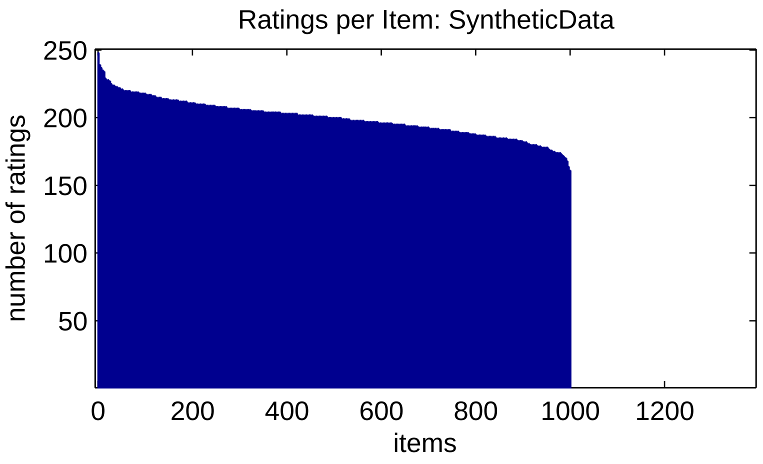

Figure 2.14: Synthetic dataset configuration 1: The ratings per item are nearly equally distributed. This property comes from the fact that the observed ratings were drawn randomly from the full matrix $\mathbf{R}$.

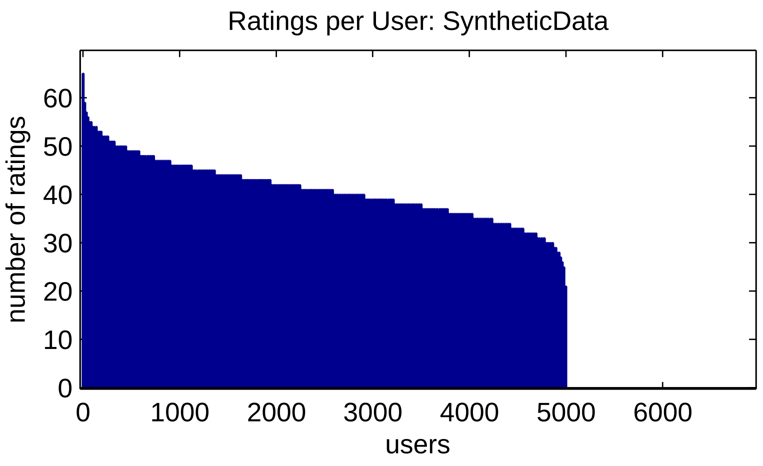

Figure 2.15: Synthetic dataset configuration 1: The ratings per user are nearly equally distributed.

2.5.3 Configuration 3

The third standard configuration uses 5,000 users and 1,000 items, which is exactly the same as for the first two configurations. The global rating mean is 3.54, the rating observation noise is $\sigma = 0.7$ and the ratings are rounded. In contrast to the previous settings the fillrate is much higher, in this configuration a fillrate of 16% is used. The rating histogram is shown in Figure 2.17.

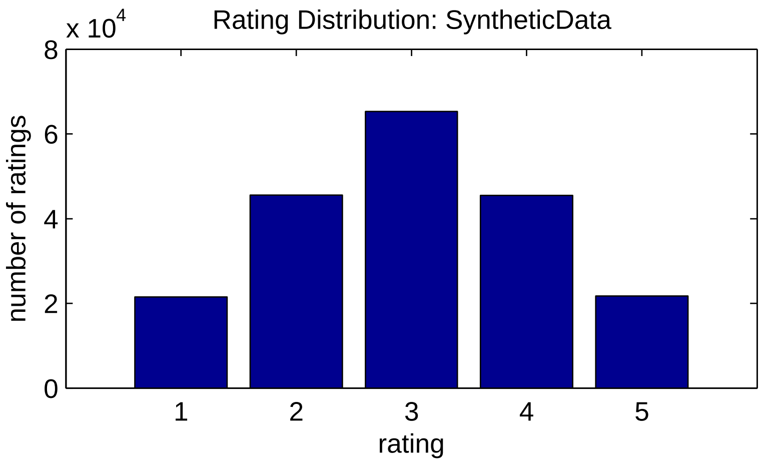

Figure 2.16: Synthetic dataset configuration 2: The mean rating is 3. In contrast to configuration 1 the ratings have been rounded.

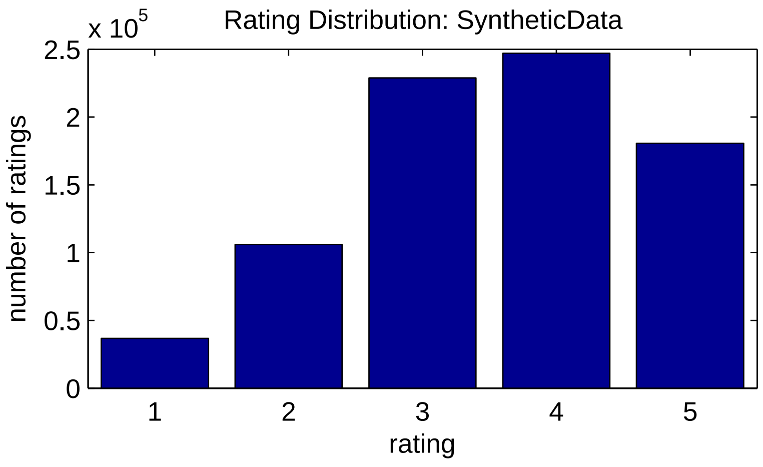

Figure 2.17: Synthetic dataset configuration 3: Due to the shift of the rating mean to 3.54, the 3 and 4 star rating is now very popular.

Chapter 3

Evaluation

Contents

3.1 Overview

The fields of application of recommender systems are very widespread. So researchers developed a variety of error measures. Herlocker et al. [16] grouped them into the following three main groups:

- **Predictive Accuracy Measures**
- **Classification Accuracy Measures**
- **Ranking Accuracy Measures**

In each group there exist a lot of different error measures. This amount of different error measures makes it very hard to compare results. Another problem here is that for recommender systems people typically use a fixed probe set instead of cross fold validation, due to the size of real world datasets. Therefore the results become strongly dependent of the way of choosing the probe subset.

The Netflix competition boosted recommender systems research. By means of fixing a probe set and using the RMSE as an error measure the results got comparable.

A objective way of evaluating recommender systems is absolutely necessary. Unfortunately, there is no universal error measure which evaluates every aspect of a recommender system. For designing and tuning algorithms it is obligatory to have an objective way for comparison. In the case of recommender systems it is very tricky to find a good universal error measure [7].

3.2 Predictive Accuracy Measures

Predictive accuracy measures use the difference between the predicted and the real rating. Such error measures are typically used for regression problems. Examples are:

- **MAE** Mean Absolute Error
- **NMAE** Normalized Mean Absolute Error
- **MSE** Mean Square Error
- **RMSE** Root Mean Square Error

3.2.1 Mean Absolute Error - MAE

The MAE is a classical accuracy measure and frequently used for regression problems. Small and big errors influence the MSE equally. The MAE is given by

$$E_{MAE} = \frac{1}{|\mathcal{P}|} \sum_{(u,i)\in\mathcal{P}} |r_{ui} - \hat{r}_{ui}| \,, \qquad (3.1)$$

where $\mathcal{P}$ stands for the probe set, r_{ui} for the real rating and $\hat{r}_{ui}$ for the predicted rating.

3.2.2 Normalized Mean Absolute Error - NMAE

Goldberg at al.[13] proposed to normalize the MAE to the rating scale. This makes errors from datasets with different rating scales better compareable.

The normalized MAE is given by

$$E_{NMAE} = \frac{E_{MAE}}{r_{max} - r_{min}} = \frac{1}{|\mathcal{P}|} \sum_{(u,i) \in \mathcal{P}} \frac{|r_{ui} - \hat{r}_{ui}|}{r_{max} - r_{min}}, \tag{3.2}$$

where r_{max} denotes the highest possible rating and r_{min} the lowest.

3.2.3 Mean Square Error - MSE

The MSE is an accuracy measure and often used in literature for regression problems. In contrast to the MAE larger errors contribute more

$$E_{MSE} = \frac{1}{|\mathcal{P}|} \sum_{(u,i) \in \mathcal{P}} (r_{ui} - \hat{r}_{ui})^2. \tag{3.3}$$

3.2.4 Root Mean Square Error - RMSE

The RMSE is the square root of the MSE

$$E_{RMSE} = \sqrt{E_{MSE}} = \sqrt{\frac{1}{|\mathcal{P}|} \sum_{(u,i) \in \mathcal{P}} (r_{ui} - \hat{r}_{ui})^2}. \tag{3.4}$$

3.3 Classification Accuracy Measures

A recommender system could not only be seen as a regression problem. It could also be seen as a classification problem. The goal is to correctly classify the interesting versus the uninteresting items. For example on a five star rating scale this could mean to classify the 4 and 5 star rated items as interesting and the rest as uninteresting.

3.3.1 Area Under Curve - AUC

The receiver operating characteristics (ROC) curve was introduced by Hanley and McNeil [15] in the context of signal detection theory. Over time the ROC curve had become a standard tool for measuring the performance of a classifier. In order to make this metric easier to compare, it is compressed into a single number, namely the area under curve (AUC).

We calculate the AUC as described in [8]. For the evaluation of a CF algorithm, one can simply calculate the AUC per user and average over the users. In the following equation I_u^H denotes the set of items with a high target rating from user u (4 and 5 stars). While I_u^L stands for the set of items with a low target rating from user u. So these two sets are disjoint $I_u^H \cap I_u^L = \emptyset$ and their union is the set of all rated items $I_u^H \cup I_u^L = I_u$ by user u

$$E_{AUC} = \frac{1}{|U|} \sum_{u \in U} \sum_{\hat{i} \in I_u^H} \sum_{\bar{i} \in I_u^L} \frac{1_{\hat{r}_{u\hat{i}} > \hat{r}_{u\bar{i}}}}{|I_u^H| \cdot |I_u^L|}. \tag{3.5}$$

3.4 Rank Accuracy Measures

Rank accuracy measures are used to quantify the quality of a sorting. Here a recommender system predicts recommendation scores with the items getting sorted based on these scores. Some of the error measures in this category assume a binary view on the data, as for the classification accuracy measures.

3.4.1 Normalized Cumulative Gain - NCG

The NCG is often used to measure the quality of list of search results. This error measure is often used for search engines. Let us assume a binary classification of all items in interesting and uninteresting. The interesting items have a relevance score of $rel_i = 1$ and the uninteresting items $rel_i = 0$. In order to calculate $E_{CG}(p)$ the recommender system predicts recommendation scores for a list of items. The list gets sorted while the relevance scores of the first p items get summed up

$$E_{CG}(p) = \sum_{i=1}^{p} rel_i. \tag{3.6}$$

In order to normalize this error measure $E_{CG}(p)$ is divided by $E_{ICG}(p)$ which is the sum of the first p relevance scores based on a ideal sorting

$$E_{NCG}(p) = \frac{E_{CG}(p)}{E_{ICG}(p)}. \tag{3.7}$$

Hence, the best value for the $E_{NCG}(p)$ is 1 and 0 is the worst. The NCG is not restricted to binary relevance scores, it is also possible to use ratings as relevance scores. Another interesting property of this error measure is, that only ranking the first elements influences the error. This is totally different to the predictive accuracy measures where every error counts the same.

3.4.2 Normalized Discounted Cumulative Gain - NDCG

The NDCG introduces a discount factor, so a correct sorting at the highest places is more importend. As for the NCG the last places (higher than p) do not influence the error

$$E_{DCG}(p) = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{log_2(i)}, \tag{3.8}$$

$$E_{NDCG}(p) = \frac{E_{DCG}(p)}{E_{IDCG}(p)}. \tag{3.9}$$

3.4.3 Average Rank - AR

The average rank works only on binary data. As for the other ranking errors the list of items is sorted according to the predicted ratings $\hat{r}_{ui}$, where $rank(\hat{r}_{ui})$ denotes the position (ranking) of the predicted rating within the list of all predicted ratings for the user u. The E_{AR} is now the average ranking of items with an high rating

$$E_{AR} = \frac{1}{|U|} \sum_{u \in U} \left(\frac{1}{|I_u^H|} \sum_{i \in I_u^H} \text{rank}(\hat{r}_{ui}) \right), \tag{3.10}$$

such that lower values of E_{AR} indicate better performance.

3.5 Conclusion

In this chapter we introduced several popular error measures for collaborative filtering algorithms. These error measures were grouped into predictive accuracy, classification and ranking error measures. Throughout the rest of this work we focus on RMSE, MAE, AR and AUC. We will report errors for all these error measures for every experiment. The main intention of this work is not to achieve new benchmark results on a given dataset. Instead we are interested in the relationships between the error measures which will be investigated in Chapter 6 empirically.

Chapter 4

Algorithms

Contents

4.1 Overview

Collaborative filtering algorithms can be categorized into memory based and model based [6]. Memory based algorithms need all the data in the main memory for making predictions. Typical examples in this category are nearest neighbor algorithms. By contrast, model based methods learn a user and item model, so these methods do not need access to the data for generating predictions. Good examples are factor models. In practical implementations pure memory based methods are rarely used because of scalability problems. For example the correlations for neighborhood based methods are typically precomputed, so this method can become a hybrid algorithm in real world applications.

4.2 K-Nearest Neighbor - KNN

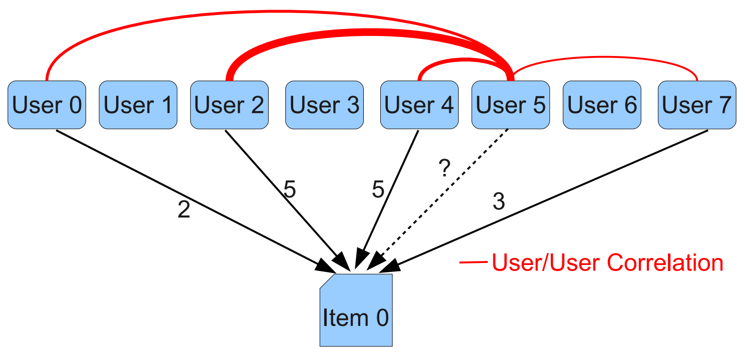

Figure 4.1: This figure visualizes a user based KNN. The thickness of red lines between users indicate the correlations between them, which are calculated based on past ratings. Thick red lines stand for strong correlations, while thinner red lines denote weaker correlations. The goal in this example is to predict a rating of user 5 for item 0. The unpersonalized approach would be to take the mean rating of all other users. In order to personalize the rating on item 0 for user 5, we select the K most similar users and weight their rating based on the correlation to user 5. So the unpersonalized average over all users changes to a weighted average over the most similar users.

Nearest neighbor algorithms are very popular for collaborative filtering, and were one of earliest published. In 1994 Resnick et al. [26] published a user based KNN for filtering netnews, which become very popular. KNN algorithms are the best example for a memory based method. The basic idea is to calculate correlations between users or between items and to make predictions based on the ratings of the top K correlating users or items.

4.2.1 User KNN

The user KNN is based on user/user correlations. A example is shown in Figure 4.1. The needed similarities between two users u and v are computed based on the set of items that both users have rated $I_{uv} = I_u \cap I_v$. The typical choice for the correlation

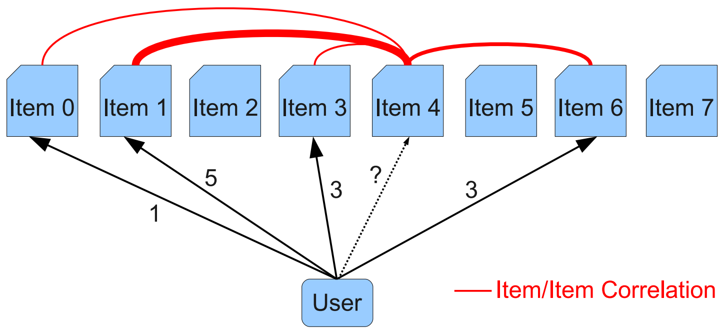

Figure 4.2: This example visualizes a item correlation based KNN. It is basically a flipped version of the example in Figure 4.1. For item KNN a similarity between the items is used instead of user similarities. Thus red lines denote similarities between the items. Thicker lines denote strong correlations and thinner lines stand for weak correlations. In order to predict the rating for item 4, one has to simply take the K ratings for items which are most similar to item 4. Finally a weighted average based on the K most similar ratings is calculated.

between two users is the Pearson correlation ρ_{uv}, which is given by

$$\rho_{uv} = \frac{\frac{1}{|I_{uv}|-1} \sum_{i \in I_{uv}} (r_{ui} - \mu_u)(r_{vi} - \mu_v)}{\sqrt{\frac{1}{|I_{uv}|-1} \sum_{i \in I_{uv}} (r_{ui} - \mu_u)^2} \sqrt{\frac{1}{|I_{uv}|-1} \sum_{i \in I_{uv}} (r_{vi} - \mu_v)^2}}, \tag{4.1}$$

where

$$\mu_u = \frac{1}{|I_{uv}|} \sum_{i \in I_{uv}} r_{ui} \tag{4.2}$$

$$\mu_v = \frac{1}{|I_{uv}|} \sum_{i \in I_{uv}} r_{vi}. \tag{4.3}$$

In order to compute the predicted rating $\hat{r}_{ui}$ for user u on item i, one has to select the K most similar users $U_i(u;K)$ who rated item i. This set of similar users is a subset of all users $U_i(u;K) \subset U_i$ who have rated item i, where $\forall_{v_1 \in U_i(u;K)} \forall_{v_2 \in (U_i - U_i(u;K))} : \rho_{v_1 u} \geq \rho_{v_2 u}$. The basic KNN version would be to simply average the ratings of the similar user set, which leads to:

$$\hat{r}_{ui} = \frac{1}{|U_i(u;K)|} \sum_{v \in U_i(u;K)} r_{vi} \tag{4.4}$$

A more sophisticated way of calculating this KNN is to calculate a weighted average, where the ratings are weighted with the user correlation.

$$\hat{r}_{ui} = \frac{\sum_{v \in U_i(u;K)} \rho_{uv} r_{vi}}{\sum_{v \in U_i(u;K)} |\rho_{uv}|} \tag{4.5}$$

If we additionally account for the users rating mean μ_u, we end up with the user based KNN described in [26]:

$$\hat{r}_{ui} = \mu_u + \frac{\sum_{v \in U_i(u;K)} \rho_{uv}(r_{vi} - \mu_v)}{\sum_{v \in U_i(u;K)} |\rho_{uv}|} \tag{4.6}$$

4.2.2 Item KNN

A lot of collaborative filtering datasets, and all of the ones used within this work, have more users than items. This means that the average item has more ratings than the average user, so item/item correlations are typically better defined. For this reason, item based KNN typically delivers much better results. In Figure 4.2 there is a visualization of an item correlation based KNN.

The rating prediction for item based KNN is given by:

$$\hat{r}_{ui} = \frac{\sum_{j \in I_u(i;K)} \rho_{ij} r_{uj}}{\sum_{j \in I_u(i;K)} |\rho_{ij}|} \tag{4.7}$$

Where $I_u(i;K) \subset I_u$ is the set of most similar items to item i, which were rated by user u. The similarity between two items i and j is measured in terms of the Pearson correlation ρ_{ij}, which is defined on the subset of ratings from users who rated both items $U_{ij} = U_i \cap U_j$:

$$\rho_{ij} = \frac{\frac{1}{|U_{ij}|-1} \sum_{u \in U_{ij}} (r_{ui} - \mu_i)(r_{uj} - \mu_j)}{\sqrt{\frac{1}{|U_{ij}|-1} \sum_{u \in U_{ij}} (r_{ui} - \mu_i)^2} \sqrt{\frac{1}{|U_{ij}|-1} \sum_{u \in U_{ij}} (r_{uj} - \mu_j)^2}}, \tag{4.8}$$

where

$$\mu_i = \frac{1}{|U_{ij}|} \sum_{u \in U_{ij}} r_{ui} \tag{4.9}$$

$$\mu_j = \frac{1}{|U_{ij}|} \sum_{u \in U_{ij}} r_{uj}. \tag{4.10}$$

4.2.3 Correlation Shrinkage

A problem of the user and item correlations, as described above, is that not all of them can be calculated with the same confidence level. The move from user to item correlations helps on datasets with fewer items to get better defined correlations on average. Still, there is the problem that some correlations are based on thousands of ratings, while others on less than ten. An easy and effective solution was proposed by Bell and Koren [4]. They proposed to shrink the correlations towards zero based on the number of ratings used to estimate the correlation. The “shrinked” correlation c_{ij} between item i and j is given by

$$c_{ij} = \frac{|U_{ij}| \cdot \rho_{ij}}{|U_{ij}| + \alpha}, \tag{4.11}$$

where α is a meta parameter.

Thus, when the number of users $|U_{ij}|$ who rated item i and j is high compared to α, the raw correlation ρ_{ij} stays nearly unchanged. If the correlation is badly defined, $|U_{ij}|$ is small compared to α, so ρ_{ij} gets shrunken towards 0.

We can directly use this shrunken correlation c_{ij} for the item KNN for Equation 4.7, which results in:

$$\hat{r}_{ui} = \frac{\sum_{j \in I_u(i;K)} c_{ij} r_{uj}}{\sum_{j \in I_u(i;K)} |c_{ij}|} \tag{4.12}$$

4.2.4 Nonlinear Correlation Rescaling

During the Netflix competition we discovered that weighting the neighboring ratings with raw or shrunken correlation is not optimal. We got improved results by using a sigmoid mapping function σ, which maps the correlations to weights better suited for calculating weighted averages [32]. The rescaled correlation $\hat{c}_{ij}$ between item i and j is given by

$$\hat{c}_{ij} = \sigma\left(\delta \cdot c_{ij} + \gamma\right), \tag{4.13}$$

where δ, γ are meta parameters, and

$$\sigma(x) = \frac{1}{1 - e^{-x}} \tag{4.14}$$

is a sigmoidal nonlinearity.

The mapped correlations can now be used in Equation 4.12, which leads to:

$$\hat{r}_{ui} = \frac{\sum_{j \in I_u(i;K)} \hat{c}_{ij} r_{uj}}{\sum_{j \in I_u(i;K)} |\hat{c}_{ij}|} \tag{4.15}$$

Throughout the following work we refer to the item KNN with nonlinear correlation rescaling as “KNN V2”.

4.2.5 Preprocessing

In the Netflix competition it has been shown that preprocessing is very useful for reducing the error in terms of the RMSE. Already substraction of user and item biases reduces the RMSE significantly. Robert Bell and Yehuda Koren reported 10 global effects for preprocessing, which drastically improved the RMSE [4]. We described 4 new global effects [32] for the Netflix progress prize 2008 and 2 new for the grand prize [33].

Restricted Boltzmann Machines as described in [28] where also shown to achieve great results for preprocessing.

4.3 Singular Value Decomposition - SVD

The singular value decomposition is well known in linear algebra and states that an arbitrary matrix $\mathbf{R}$ with real or complex entries can be decomposed into a product of three matrices $\mathbf{A\Sigma B}^T$ with $\mathbf{\Sigma}$ being a diagonal matrix. The SVD can be used for calculating the Pseudoinverse, solving homogeneous linear equations and low rank approximations. There exist a lot of theory about SVD, but most of it can not be used directly for collaborative filtering. In the collaborative filtering context the matrix $\mathbf{R}$ is sparse, meaning that most of the elements are unknown. The basic idea of SVD to decompose the matrix $\mathbf{R}$ can be still applied. However, many of the nice properties of standard SVD do not hold anymore.

The basic idea is to represent a user by an user feature vector $\mathbf{a}_u$ and an item by an item feature vector $\mathbf{b}_i$. This means the users are represented by a $|U| \times N$ matrix $\mathbf{A}$, where N is the number of factors used to represent a user. The items are represented by a $|I| \times N$ matrix $\mathbf{B}$. The final goal is that the product of the matrices $\mathbf{A}\mathbf{B}^T$ approximates the known ratings of $\mathbf{R}$.

The goal is to $min_{\mathbf{A},\mathbf{B}}\|\mathbf{R} - \mathbf{A}\mathbf{B}^T\|_R^2$, where the norm $\|\cdot\|_R^2 = \sum_{(u,i)\in\mathcal{L}} r_{ui}^2$ is only defined on the known ratings of $\mathbf{R}$. In order to account for the different number of known ratings for users and items, regularizing the matrix decomposition is very important. This leads to the following error function to be minimized:

$$E(\mathbf{A},\mathbf{B}) = \|\mathbf{R} - \mathbf{A}\mathbf{B}^T\|_R^2 + \lambda(\|\mathbf{A}\|_F^2 + \|\mathbf{B}\|_F^2) \tag{4.16}$$

In the equation above $\|\cdot\|_F^2$ denotes the squared Frobenius norm and λ is the meta parameter for controlling the L2 regularization. It was found that introducing a global bias c, user bias d_u and item bias e_i helps to improve the results [24]. A rating prediction is then given by:

$$\hat{r}_{ui} = c + d_u + e_i + \mathbf{a}_u^T\mathbf{b}_i \tag{4.17}$$

The error function to be minimized can be written as:

$$E(\mathbf{A},\mathbf{B},c,\mathbf{d},\mathbf{e}) = \sum_{(u,i)\in\mathcal{L}} (r_{ui} - \hat{r}_{ui})^2 + \lambda(\|\mathbf{A}\|_F^2 + \|\mathbf{B}\|_F^2 + \|\mathbf{d}\|_F^2 + \|\mathbf{e}\|_F^2) \tag{4.18}$$

Where $\mathbf{d}$ is the vector containing the user biases and $\mathbf{e}$ the item biases.

In order to estimate all parameters in the above model one has to precalculate the global mean c and learn all remaining parameters using stochastic gradient descent. To do this, all parameters are initialized with a random value drawn from a uniform distribution around zero $[-0.001, 0.001]$. The full details of the training process can be found in Algorithm 1.

4.3.1 Remarks

Training a SVD using stochastic gradient descent is obviously not the only training method, batch and mini batch updates are also commonly used. Full batches have the disadvantage that a lot of epochs are needed due to rare parameter updates. Mini batches are a tradeoff between full batches and updates after every training sample

```
1  Initialize $c$ to the global mean.
2  Initialize $\mathbf{A}$, $\mathbf{B}$, $\mathbf{d}$ and $\mathbf{e}$ from a uniform distribution $[-0.001, 0.001]$.
3  repeat
4      for $(u, i) \in \mathcal{T}$ do
5          $\hat{e}_{ui} \leftarrow r_{ui} - \hat{r}_{ui}$
6          $d_u \leftarrow d_u + \eta(\hat{e}_{ui} - \lambda d_u)$
7          $e_i \leftarrow e_i + \eta(\hat{e}_{ui} - \lambda d_u)$
8          for $n = 1$ to $N$ do
9              $\tilde{a} \leftarrow a_{un}$
10             $a_{un} \leftarrow a_{un} + \eta(\hat{e}_{ui} b_{in} - \lambda a_{un})$
11             $b_{in} \leftarrow b_{in} + \eta(\hat{e}_{ui} \tilde{a} - \lambda b_{in})$
12 until error is minimal on $\mathcal{P}$
```

Algorithm 1: Training a SVD with stochastic gradient descent. In the beginning c is initialized to the global mean. All other parameters are initialized randomly around zero. Then stochastic gradient descent is performed, until a minimum on the probe set is reached.

(stochastic update). Furthermore the mini batch update has the pleasant property that it is easy to parallelize, because the training samples can be calculated within a mini batch in parallel. During the Netflix competition we discovered that larger mini batches lead to inferior results and increase the number of needed epochs, which nearly nullifies the speedup through parallelization. Hence we always used stochastic gradient descent.

Another very popular method for training SVDs is to use alternating least squares [4]. The basic idea is to fix the user weights and calculate the item dependent weights, which reduces the problem to solving a system of linear equations. Then the item weights are fixed and the user weights are calculated. This process is repeated until it converges. In the paper the authors also describe a way to modify simple least squares to restrict all parameters to be non negative. In our experience alternating least squares produce inferior results compared to simple stochastic gradient decent. The biggest problem of this method is that the computation time rises quadratically with the feature size N.

4.4 Asymmetric Factor Model - AFM

In some datasets the ratings of the sparse rating matrix $\mathbf{R}$, are not missing at random. When the user has the free choice which item he is going to rate, then already the selection of an item contains valueable information. We will call this information, which is contained already in the selction of an item for rating, the selection bias.

The asymmetric factor model was first described by Arkadiusz Paterek in [24]. The basic idea is to model the selection bias. There exist two distinct sets of item features, which are used $\mathbf{b}_i, \mathbf{f}_i \in \mathcal{R}^N$. The SVD models a user directly via a user feature vector $\mathbf{a}_u$. By contrast, an AFM models a user as the bag of the rated items. The predicted rating of user u for item i is given by

$$\hat{r}_{ui} = \mathbf{b}_i^T \left(\frac{1}{|I_u|} \sum_{j \in I_u} \mathbf{f}_j \right) . \tag{4.19}$$

In the Netflix competition it was found that user and item biases help to improve the results:

$$\hat{r}_{ui} = c + d_u + e_i + \mathbf{b}_i^T \left(\frac{1}{|I_u|} \sum_{j \in I_u} \mathbf{f}_j \right) , \tag{4.20}$$

where c is the global bias, d_u the user bias and e_i the item bias. Using a quadratic error leads to the following regularized error function:

$$E(\mathbf{B}, \mathbf{F}, c, \mathbf{d}, \mathbf{e}) = \sum_{(u,i) \in \mathcal{L}} (r_{ui} - \hat{r}_{ui})^2 + \lambda(\|\mathbf{B}\|_F^2 + \|\mathbf{F}\|_F^2 + \|\mathbf{d}\|_F^2 + \|\mathbf{e}\|_F^2) \tag{4.21}$$

The parameters of the AFM can be learned using stochastic gradient descent, which leads to the algorithm described in Algorithm 2. If the average user has rated lots of items, the pure stochastic update is very time consuming and it is better to do a user wise batch update for the asymmetric item features $\mathbf{f}_i$ as in Algorithm 3.

4.4.1 Remarks

Paterek reported a variation of this model with only one set of item features $\mathbf{b}_i = \mathbf{f}_i$. In the Netflix competition this modification made the individual RMSE always worse,

```
1  Initialize $c$ to the global mean.
2  Initialize $\mathbf{B}$, $\mathbf{F}$, $\mathbf{d}$ and $\mathbf{e}$ from a uniform distribution $[-0.001, 0.001]$.
3  repeat
4      for $(u, i) \in \mathcal{T}$ do
5          $\hat{e}_{ui} \leftarrow r_{ui} - \hat{r}_{ui}$
6          $d_u \leftarrow d_u + \eta(\hat{e}_{ui} - \lambda d_u)$
7          $e_i \leftarrow e_i + \eta(\hat{e}_{ui} - \lambda d_u)$
8          for $n = 1$ to $N$ do
9              $\tilde{b} \leftarrow b_{in}$
10             $b_{in} \leftarrow b_{in} + \eta\left(\hat{e}_{ui}\left(\frac{1}{|I_u|}\sum_{j \in I_u} f_{jn}\right) - \lambda b_{in}\right)$
11             for $j \in I_u$ do
12                 $f_{jn} \leftarrow f_{jn} + \eta\left(\hat{e}_{ui}\frac{\tilde{b}}{|I_u|} - \lambda f_{jn}\right)$
13 until error is minimal on $\mathcal{P}$
```

Algorithm 2: The training of a AFM with pure stochastic gradient descent. For datasets with a lot of ratings per user, the most inner loop becomes very time consuming. A solution for this problem can be found in Algorithm 3.

but helped when different predictions were combined to increase prediction accuracy.

4.5 SVD++

The SVD++ combines the ideas of SVD and AFM, which Yehuda Koren described in [19]. Combining those ideas leads to the following prediction formula:

$$\hat{r}_{ui} = c + d_u + e_i + \mathbf{b}_i^T\left(\mathbf{a}_u + \frac{1}{|I_u|}\sum_{j \in I_u} \mathbf{f}_j\right), \tag{4.22}$$

where c is a global bias, d_u a user bias and e_i an item dependent bias. The vectors $\mathbf{b}_i$ and $\mathbf{f}_j$ are item dependent feature vectors, while $\mathbf{a}_u$ is a user feature vector. Using the prediction formula from Equation 4.22 and a quadratic loss function leads to the following error function:

$$E(\mathbf{A}, \mathbf{B}, \mathbf{F}, c, \mathbf{d}, \mathbf{e}) = \sum_{(u,i) \in \mathcal{L}} (r_{ui} - \hat{r}_{ui})^2 + \lambda(\|\mathbf{A}\|_F^2 + \|\mathbf{B}\|_F^2 + \|\mathbf{F}\|_F^2 + \|\mathbf{d}\|_F^2 + \|\mathbf{e}\|_F^2) \tag{4.23}$$

```
1  Initialize $c$ to the global mean.
2  Initialize $\mathbf{B}$, $\mathbf{F}$, $\mathbf{d}$ and $\mathbf{e}$ from a uniform distribution $[-0.001, 0.001]$.
3  repeat
4      for $u \in U$ do
5          $\tilde{\mathbf{f}} \leftarrow \left(\frac{1}{|I_u|} \sum_{j \in I_u} \mathbf{f}_j\right)$
6          for $i \in I_u$ do
7              $\hat{e}_{ui} \leftarrow r_{ui} - \hat{r}_{ui}$
8              $d_u \leftarrow d_u + \eta(\hat{e}_{ui} - \lambda d_u)$
9              $e_i \leftarrow e_i + \eta(\hat{e}_{ui} - \lambda d_u)$
10             $\hat{\mathbf{f}} \leftarrow \mathbf{0}$
11             for $n = 1$ to $N$ do
12                 $\hat{f}_n \leftarrow \hat{f}_n + \hat{e}_{ui} b_{in}$
13                 $b_{in} \leftarrow b_{in} + \eta\left(\hat{e}_{ui} - \lambda b_{in}\right)$
14         for $i \in I_u$ do
15             $f_{in} \leftarrow f_{in} + \eta\left(\frac{\hat{f}_{in}}{|I_u|} - \lambda f_{in}\right)$
16 until error is minimal on $\mathcal{P}$
```

Algorithm 3: This algorithm block shows the training of a AFM with a user wise batch update for the asymmetric item features $\mathbf{f}_j$. All other parameters are updated stochastically. Instead of training all samples randomly as in Algorithm 2 one iterates first over all users u and then over all rated items for this user I_u. Due to the batch update of the asymmetric item features $\mathbf{f}_j$ it is possible to precompute the virtual user feature vector $\tilde{\mathbf{f}}$. Then one iterates over all rated items of this user and accumulates a batch update in $\hat{\mathbf{f}}$ and updates the asymmetric item features $\mathbf{f}_j$ afterwards.

The training of the parameters can be done with gradient descent in the same way as for the AFM, which leads to Algorithm 4.

4.6 RankSVD

Most algorithms see the recommendation task as a rating prediction task. Items can be sorted according to their predicted rating and the items with the highest predicted ratings are recommended to the user. Thus, for the standard recommendation task the main interest lies in sorting items. Predicting ratings is the most popular way to get a sorting, but it is not the only one. Pessiot et al. [25] showed a efficient way to directly learn a ranking or sorting.

```
1  Initialize $c$ to the global mean.
2  Initialize $\mathbf{A}$, $\mathbf{B}$, $\mathbf{F}$, $\mathbf{d}$ and $\mathbf{e}$ from a uniform distribution $[-0.001, 0.001]$.
3  repeat
4      for $u \in U$ do
5          $\tilde{\mathbf{f}} \leftarrow \left(\frac{1}{|I_u|}\sum_{j\in I_u}\mathbf{f}_j\right)$
6          for $i \in I_u$ do
7              $\hat{e}_{ui} \leftarrow r_{ui} - \hat{r}_{ui}$
8              $d_u \leftarrow d_u + \eta(\hat{e}_{ui} - \lambda d_u)$
9              $e_i \leftarrow e_i + \eta(\hat{e}_{ui} - \lambda d_u)$
10             $\hat{\mathbf{f}} \leftarrow \mathbf{0}$
11             for $n = 1$ to $N$ do
12                 $\tilde{b} \leftarrow b_{in}$
13                 $\hat{f}_n \leftarrow \hat{f}_n + \hat{e}_{ui}b_{in}$
14                 $b_{in} \leftarrow b_{in} + \eta\left(\hat{e}_{ui} - \lambda b_{in}\right)$
15                 $a_{un} \leftarrow a_{un} + \eta(\hat{e}_{ui}b_{in} - \lambda a_{un})$
16         for $i \in I_u$ do
17             $f_{in} \leftarrow f_{in} + \eta\left(\frac{\hat{f}_{in}}{|I_u|} - \lambda f_{in}\right)$
18 until error is minimal on $\mathcal{P}$
```

Algorithm 4: This algorithm block describes the trainings process of a SVD++ model. The user and item features are updated using a update after every training example, while the asymmetric item features $\mathbf{f}_j$ are getting an user wise batch update.

The basic idea is similar to the SVD. A user u is represented via user feature vector $\mathbf{a}_u$ and the a item i with the vector $\mathbf{b}_i$. In contrast to the SVD, where a dot product of user and item feature vectors is used to approximate the rating matrix, the RankSVD predicts a ranking score. Based on this score it is possible to sort the items accordingly and recommend the highest ranked items.

In order to learn a ranking a exponential loss function is used, which leads to the following error function.

$$E(\mathbf{A}, \mathbf{B}) = \sum_{u\in U} \sum_{\substack{i,j\in I_u \\ r_{ui}<r_{uj}}} e^{\mathbf{a}_u^T\mathbf{b}_i - \mathbf{a}_u^T\mathbf{b}_j} + \lambda(\|\mathbf{A}\|_F^2 + \|\mathbf{B}\|_F^2) \tag{4.24}$$

The error function sums the errors over all users and all rating combinations of this user. Ranking scores which lead to a wrong pairwise ranking get a high error. The wronger the predicted ranking score, the higher the error. The same is true for correct rankings. The bigger the margin the smaller the contribution to the overall error. So the error function is not only designed to lead to correct rankings it is also designed to improve the margin.

In the paper Pessiot et al. suggested to train **A**, **B** one after another. First **A** is optimized while **B** is fixed. Then **B** is optimized while **A** is fixed. This process is repeated until convergence.

We use stochastic gradient descent, as for all the other factor models, which we found gives better results in a shorter time. Using gradient descent leads to Algorithm 5.

```
1  Initialize $\mathbf{A}$, $\mathbf{B}$ from a uniform distribution $[-0.001, 0.001]$.
2  repeat
3      for $(u, i) \in \mathcal{T}$ do
4          $j$ is selected randomly from $I_u$
5          if $r_{uj} \leq r_{ui}$ then
6              $\tilde{t} \leftarrow i$
7              $i \leftarrow j$
8              $j \leftarrow \tilde{t}$
9          $\hat{e}_{ui} \leftarrow e^{\mathbf{a}_u^T \mathbf{b}_i - \mathbf{a}_u^T \mathbf{b}_j}$  for $n = 1$ to $N$ do
10             $\tilde{a} \leftarrow a_{un}$
11             $a_{un} \leftarrow a_{un} - \eta(\hat{e}_{ui}(b_{in} - b_{jn}) + \lambda a_{un})$
12             $b_{in} \leftarrow b_{in} - \eta(\hat{e}_{ui}\tilde{a} + \lambda b_{in})$
13             $b_{jn} \leftarrow b_{in} - \eta(\hat{e}_{ui}(-\tilde{a}) + \lambda b_{jn})$
14 until error is minimal on $\mathcal{P}$
```

Algorithm 5: Training a RankSVD with stochastic gradient descent. In the beginning, user and item features are initialized randomly. Then they are trained epoch wise, while each epoch runs over all available training samples $\mathcal{T}$. In the inner loop we randomly select an rated item j from the set of items I_u rated by user u. Note that this means a simplification and approximation. In order to exactly minimize the error function from Equation 4.24 one has to loop over all items I_u with an higher rating than r_{ui}. This would lead to big runtime problems for datasets with a lot of user ratings (e.g. Netflix), because of the computational time growing quadratically with the number of ratings per user. On the MovieLens 1M dataset this simplification has not changed the accuracy, but it made it possible to run the algorithm on big datasets like Netflix.

4.6.1 Remarks

A very good property of the RankSVD is the invariance to monotonic transformations of user ratings. The RankSVD uses only pairwise comparisons of ratings per user, so user biases and monotonic transformations do not have any impact. Hence, there is no need to model these effects. In the extreme case every user can use his own rating scale. For the algorithm it makes no difference if some users use a 3 star rating scale while others use 5 or 10 stars.

Chapter 5

Parameter Tuning

Contents

5.1 Overview

Every collaborative filtering algorithm discussed within this work contains several meta parameters. These parameters are used to control the regularization and the model complexity. Some of these parameters are very sensitive, while others are not. Tuning all these parameters for every dataset by hand is a time consuming process and mostly unfeasible for real world recommender systems. Therefore, an automated meta parameter search is vital.

Direct search methods can find local minima of error functions without the need of gradients, so these are a good choice for optimizing meta parameters. As for most optimization methods there is no guarantee to find a global minimum. The solution found can strongly depend on the choice of the initial values.

In the following sections 3 direct search methods will be discussed. The goal is to minimize an error function $E(\alpha_1, \alpha_2, ..., \alpha_N)$ which depends on N meta parameters α_1 to α_N.

5.2 Random Search

Random Search is a very simple idea. First, a parameter is selected at random, then the parameter is changed randomly. In the case that the changed parameter lowered the error E, the new value of the parameter is kept, otherwise the old is restored. This procedure is repeated until convergence. A more detailed explanation can be found in Algorithm 6.

```
1  Initialize $\alpha_1, \alpha_2, ..., \alpha_N$
2  $e \leftarrow E(\alpha_1, \alpha_2, ..., \alpha_N)$
3  repeat
4      Select $i$ randomly between 1 and $N$.
5      $\tilde{\alpha}_i \sim \mathcal{N}(\alpha_i; \frac{1}{10}\max(|\alpha_i|, 1))$
6      $\tilde{e} \leftarrow E(\alpha_1, \alpha_2, ..., \alpha_{i-1}, \tilde{\alpha}_i, \alpha_{i+1}, ..., \alpha_N)$
7      if $\tilde{e} < e$ then
8          $\alpha_i \leftarrow \tilde{\alpha}_i$
9          $e \leftarrow \tilde{e}$
10 until $e$ unchanged for a long time
```

Algorithm 6: In the beginning, one has to initialize the parameters α_1 to α_N. This could be at random, but the better choice is to start with known good parameters. In general this is not possible, but for CF algorithms one has typically an idea. The next step is to evaluate the error function E with the current set of parameters. Then a parameter α_i is randomly selected. Afterwards a new value for α_i is drawn from a normal distribution centered around the old value of α_i. The standard deviation is set to $\frac{1}{10}$ of absolute value of α_i. The maximum operator is used to avoid problems around 0. With the new value $\tilde{\alpha}_i$ the error function is evaluated. In the case the error is better, the new value $\tilde{\alpha}_i$ is kept as a replacement for α_i, otherwise $\tilde{\alpha}_i$ is dropped. The process of randomly changing parameters is repeated until convergence of e, which means that e stays unchanged over a long time.

5.2.1 Remarks

In the beginning of the Netflix competition we used the above method a lot. It was very easy to implement and delivered good results. The major drawbacks of this method are that the step size is not adjusted and that only one parameter is changed at a time. Hence, the performance of this method depends on the shape of the error function.

In Section 6.7 one can find an empirical evaluation of the performance of random search and a comparison with other methods.

5.3 Coordinate Search

The basic idea of coordinate search is oriented on a “human” search behavior. The parameters are changed sequentially. If the error improved several times in one direction, the step size into this direction is increased. In the case the error gets worse, the search direction is changed and the step size is reduced. A detailed description of the algorithm can be found in Algorithm 7.

5.3.1 Remarks

This algorithm can not change the sign of a parameter. So it is important to set the correct sign initially. In the case that the sign is not known the parameter α must be decomposed into a positive part α_+ and a negative part α_-, which leads to $\alpha = \alpha_+ + \alpha_-$

5.4 Nelder Mead Algorithm

The Nelder Mead or downhill simplex algorithm is a popular optimization method which does not require gradients. The method was introduced in 1965 by John Nelder and Roger Mead [23]. My experiments are based on a more recent description which can be found in [30].

The full details of the algorithm can be found in Algorithm 8.

5.4.1 Remarks

The Nelder Mead algorithm can get stuck in flat regions of the error surface. A good solution for this problem is to construct the initial simplex again around the point where the algorithm got stuck, and start it again.

```
 1 Initialize $\alpha_1, \alpha_2, ..., \alpha_N$
 2 $\beta_1, \beta_2, ..., \beta_N \leftarrow 0.8$
 3 $\gamma_1, \gamma_2, ..., \gamma_N \leftarrow 1$
 4 $e \leftarrow E(\alpha_1, \alpha_2, ..., \alpha_N)$
 5 repeat
 6     for $i = 1$ to $N$ do
 7         for $k = 1$ to 3 do
 8             $\tilde{\alpha}_i \leftarrow \alpha_i \cdot \beta_i$
 9             $\tilde{e} \leftarrow E(\alpha_1, \alpha_2, ..., \alpha_{i-1}, \tilde{\alpha}_i, \alpha_{i+1}, ..., \alpha_N)$
10             if $\tilde{e} < e$ then
11                 $\alpha_i \leftarrow \tilde{\alpha}_i$
12                 $e \leftarrow \tilde{e}$
13                 $\gamma_i \leftarrow \gamma_i + 1$
14                 if $\gamma_i \geq 2$ then
15                     $\beta_i \leftarrow \beta_i^{1.25}$
16             else
17                 $\gamma_i \leftarrow 1$
18                 $\beta_i \leftarrow \left(\frac{1}{\beta_i}\right)^{0.8}$
19 until $e$ unchanged for a long time
```

Algorithm 7: In the beginning the parameters α_1 to α_N are set to their initial values. The parameters β_1 to β_N are the search factors and initially set to 0.8. The basic idea of the structured coordinate search is to replace the random search, with a coordinate wise structured search. Therefore, one iterates sequentially over the parameters α_1 to α_N. Then the parameter α_i is multiplied with the search factor β_i. In the case that the new parameter value improves the error, the new value is kept. Otherwise the old value is restored and the search direction is reversed. When the search in one direction is successful for more than two times then the search factor β_i is raised to the power of 1.25 in order to perform bigger steps in the successful direction.

Throughout this work the following values for the Nelder Mead meta parameters are used: $\gamma = 5$, $\beta_r = 1$, $\beta_c = \frac{1}{2}$, $\beta_e = 2$ and $\beta_s = \frac{1}{2}$

```
1  Initialize $\alpha_1, \alpha_2, ..., \alpha_N$
2  $\mathbf{x}_0 \leftarrow (\alpha_1, \alpha_2, ..., \alpha_N)^T$;                     // construct the initial simplex
3  $e_0 \leftarrow E(\mathbf{x}_0)$
4  for $i = 1$ to $N$ do
5      $\mathbf{x}_i \leftarrow (\alpha_1, \alpha_2, ..., \alpha_{i-1}, \alpha_i + \gamma \cdot \mathbf{e}_i, \alpha_{i+1}, ..., \alpha_N)^T$;   // $\mathbf{e}_i$ is a unity vector
6      $e_i \leftarrow E(\mathbf{x}_i)$
7  repeat
8      for $i = 1$ to $N$ do
9          $e_h \leftarrow \max_j(e_j)$
10         $e_s \leftarrow \max_{j \neq h}(e_j)$
11         $e_l \leftarrow \min_j(e_j)$
12         $\mathbf{c} \leftarrow \frac{1}{N} \sum_{j \neq h} \mathbf{x}_j$;                       // calculate the center
13         $\mathbf{x}_r \leftarrow \mathbf{c} + \beta_r(\mathbf{c} - \mathbf{x}_h)$;          // calculate the reflection point
14         $e_r \leftarrow E(\mathbf{x}_r)$
15         if $e_l \leq e_r < e_s$ then
16             $\mathbf{x}_h \leftarrow \mathbf{x}_r$
17         else
18             if $e_r < e_l$ then
19                 $\mathbf{x}_e \leftarrow \mathbf{c} + \beta_e(\mathbf{c} - \mathbf{x}_h)$;      // calculate the expansion point
20                 $e_e \leftarrow E(\mathbf{x}_e)$
21                 if $e_e < e_r$ then
22                     $\mathbf{x}_h \leftarrow \mathbf{x}_e$
23                 else
24                     $\mathbf{x}_h \leftarrow \mathbf{x}_r$
25             else
26                 if $e_s \leq e_r < e_h$ then
27                     $\mathbf{x}_c \leftarrow \mathbf{c} + \beta_c(\mathbf{x}_r - \mathbf{c})$;          // outside contraction
28                     $e_c \leftarrow E(\mathbf{x}_c)$
29                     $\tilde{e} \leftarrow e_r$
30                 else
31                     $\mathbf{x}_c \leftarrow \mathbf{c} + \beta_c(\mathbf{x}_h - \mathbf{c})$;           // inside contraction
32                     $e_c \leftarrow E(\mathbf{x}_c)$
33                     $\tilde{e} \leftarrow e_h$
34                 if $e_c < \tilde{e}$ then
35                     $\mathbf{x}_h \leftarrow \mathbf{x}_c$
36                 else
37                     $\forall_{j \neq l} : \mathbf{x}_j \leftarrow \mathbf{c} + \beta_s(\mathbf{x}_j - \mathbf{x}_l)$ ;              // shrinkage
38 until $min(e_0, e_1, ..., e_N)$ unchanged for a long time
39 $e_l \leftarrow \min_j(e_j)$;                              // use the best found parameters
40 $(\alpha_1, \alpha_2, ..., \alpha_N)^T \leftarrow \mathbf{x}_l$
```

Algorithm 8: This algorithm block describes the Nelder Mead algorithm as used within this work.

Chapter 6

Empirical Results

Contents

6.1 Overview

The aim of this chapter is to empirically analyze interesting properties of recommender algorithms. In Chapter 2 we presented four popular real world datasets, MovieLens1M, MovieLens10M, Jester Joke and Netflix. In Chapter 3 we gave an overview of commonly used error measures. Within this chapter of empirical results we use the MAE, RMSE, AR and AUC. In Chapter 4 we gave an overview of state of the art collaborative filtering algorithms. For the empirical evaluation we use the SVD, AFM, SVD++, RankSVD and item KNN. The KNN versions with nonlinear correlation rescaling are called KNN V2.

First we are going to investigate the importance of regularizing factor models, and investigate the influence under different error measures. In Section 6.3 we analyze the feature size of factor models and the impact on different error measures and the training time. Next we are analyzing the relationship between different error measures in Section 6.4. In Section 6.5 the influence of the fillrate of the rating matrix $\mathbf{R}$ to the errors will be investigated. The effect of observational noise is examined in Section 6.6. Finally we compare the three automatic parameter tuners introduced in Chapter 5.

6.2 Regularization of Factor Models

Within this work in all factor models users and items are represented via a feature vector of the same size. For example in a SVD model every user is represented via a N dimensional feature vector and also every item has its own N dimensional feature representation. It is not possible to use different feature sizes for every user or item. One can only use the same feature size for all users and items, but on all datasets under investigation the ratings are not equally distributed among users or items. The feature size N controls the model complexity. For users or items with many ratings a lot of features can be used. For example, using $N = 30$ is fine for users with 300 ratings, but a problem for users with only 3 ratings. L2 regularization solves this problem. We can keep the feature size fixed and change the model complexity by controlling the regularization λ. So we do not overfit the parameters on users with few ratings and can use enough features to learn complex user/item relations for users with lots of ratings.

In order to analyze the influence of L2 regularization on the RMSE, MAE, AR and AUC we have made experiments with SVD, AFM, RankSVD and SVD++ on Jester Joke, MovieLens1M and Netflix datasets.

6.2.1 SVD

We use the SVD algorithm described in Section 4.3. The feature size is fixed to $N = 30$ and the learnrate to $\eta = 0.002$. In order to limit the training time we limit the number of training epochs to 400. This means if the minimum is not reached after 400 epochs, the training will be stopped anyway. This is necessary because higher regularizations tend to increase the number of epochs needed to reach the minimum. The results can be seen in Figure 6.1.

The first and very important effect we can observe, is that the training time increases for higher regularizations. The flat regions which can be observed in the green lines come from the fact, that we limit the training epochs to 400. The other very important observation is that proper regularization is very important. For example if we look at the RMSE on the Netflix dataset, with a regularization of $\lambda = 0$ the RMSE is 0.9335, by increasing the regularization to $\lambda = 0.02$ the RMSE drops to 0.9190 and increasing the regularization further to $\lambda = 0.1$ increases the RMSE to 0.9536. So regularization has a big effect, too high or too low regularizations can make the results very bad. In Figure 6.1 we marked the optimal regularization with an red cross. So it is easier to see that the optimal regularization is different for every error measure and dataset combination.

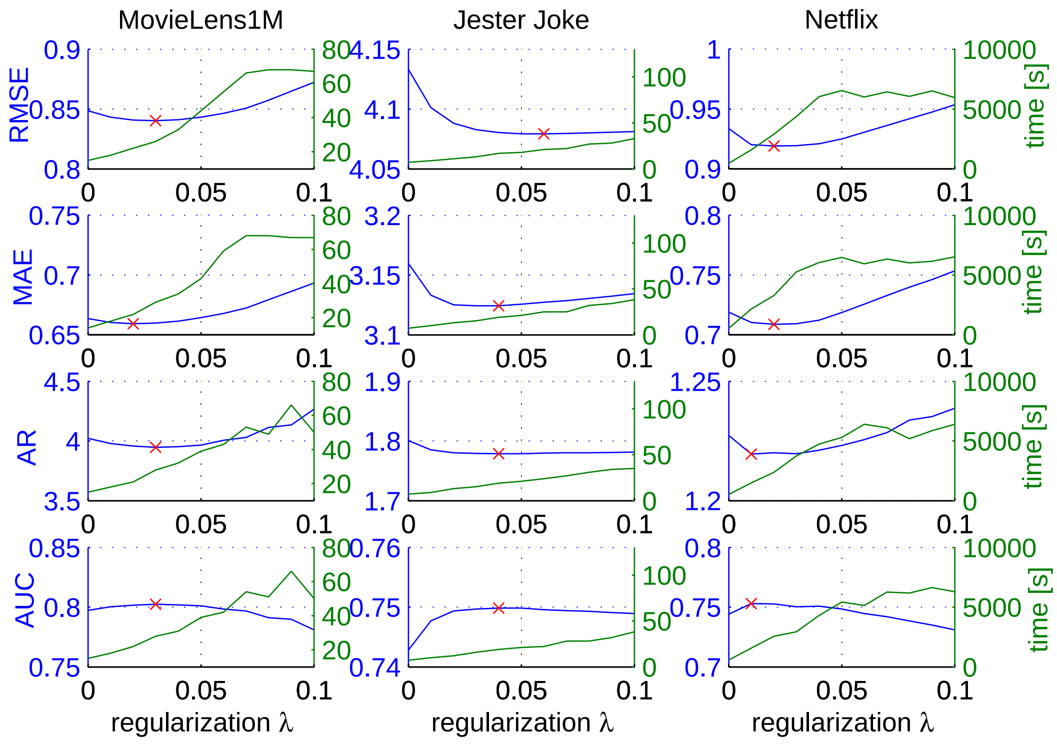

Figure 6.1: This figure shows the effect of regularization for a SVD model.

6.2.2 AFM

For this experiment we use the AFM described in Section 4.4. We used a fixed feature size of $N = 30$ and a learn rate of $\eta = 0.002$ for the Netflix and MovieLens1M datasets.

For the Jester Joke dataset we used a lower learn rate of $\eta = 0.0002$. As for the SVD we limited the number of training epochs to 400. The results are visualized in Figure 6.2.

The regularization has an big effect on the MovieLens1M and the Netflix dataset, but on the Jester Joke dataset the relative differences between high and low regularization are much smaller. This effect is probably caused by the high fillrate of the Jester Joke dataset. Another very interesting observation is, that the training time seems to behave differently compared to the SVDs in Figure 6.1, where the training time simply increased with higher regularizations. In the case of the AFM the behavior looks a bit different. When we look at the RMSE and MAE results on the Netflix dataset, we clearly see that the AFM training takes longest for the optimal regularization. For higher and lower values the training time is lower. For the Jester Joke dataset the regularization seems to have only a marginal impact on the training time.

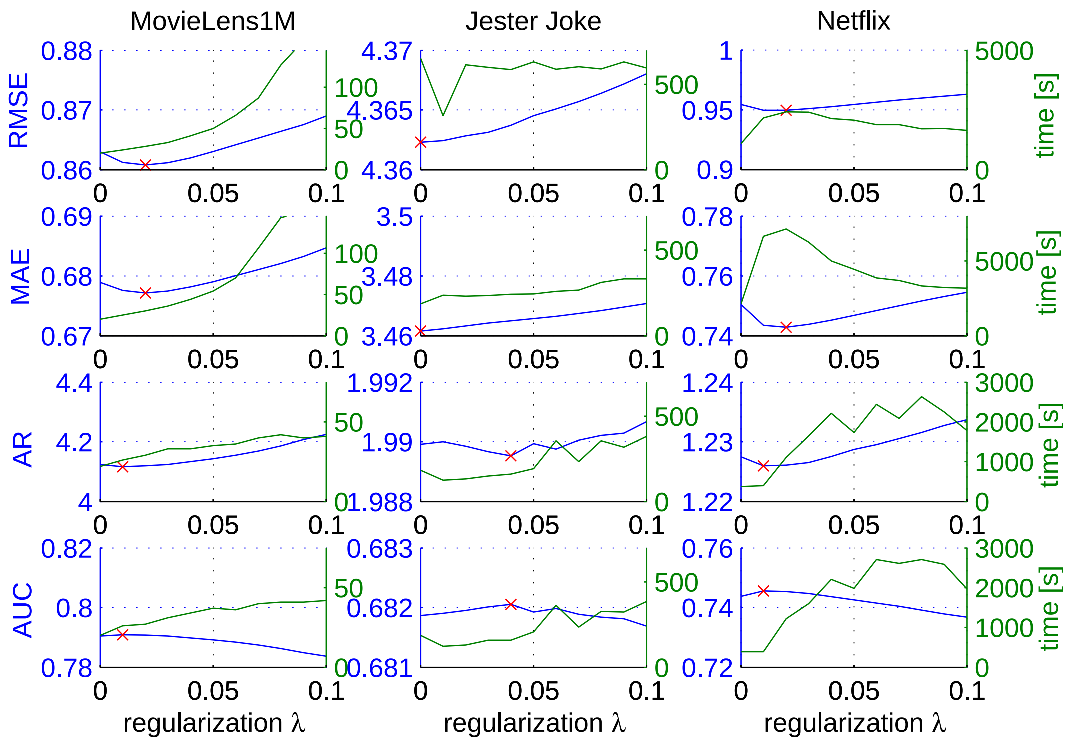

Figure 6.2: In this figure we visualize the influence of regularizing an AFM.

6.2.3 RankSVD

The RankSVD is described in Section 4.6. The feature size is fixed to $N = 30$ and the learn rate to $\eta = 0.001$. We analyzed the behavior on the MovieLens1M, Jester Joke and Netflix dataset and used the AUC and AR error measures. We do not analyze the accuracy measures RMSE and MAE, because the RankSVD can only predict rankings and not ratings. Thus it is not possible to evaluate accuracy metrics. The results can be seen in Figure 6.3.

We observe that for the MovieLens1M and the Jester Joke dataset the optimal regularization is no regularization, only for the Netflix dataset the results improve slightly by using $\lambda = 0.01$. Using high regularizations results in very bad errors. So it seems that for RankSVD the regularization is not as important as for the other factor models under investigation, and it is better to use no or low regularization.

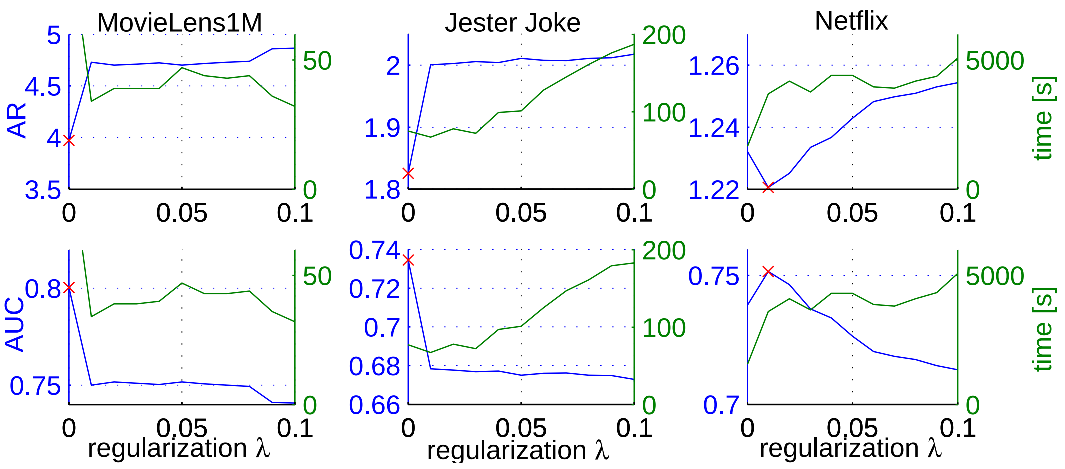

Figure 6.3: This figure visualizes the influence of regularization on RankSVD.

6.2.4 SVD++

The SVD++ is described in Section 4.5. For the following regularization experiment we used a feature size of $N = 30$ and a learn rate $\eta = 0.02$. The results of using the SVD++ on different datasets and error measures can be seen in Figure 6.4.

The first thing we noticed is the close to linear increase of the training time with higher regularizations. The flat regions of the green lines, which visualize the training time, stem from the fact that we limit the training epochs to 400. The next important point to notice is that proper regularization plays a major role. A too high or too low regularization has an negative effect on the performance. In general we observe that the optimal regularizations tend to be higher than for the SVD, AFM and RankSVD.

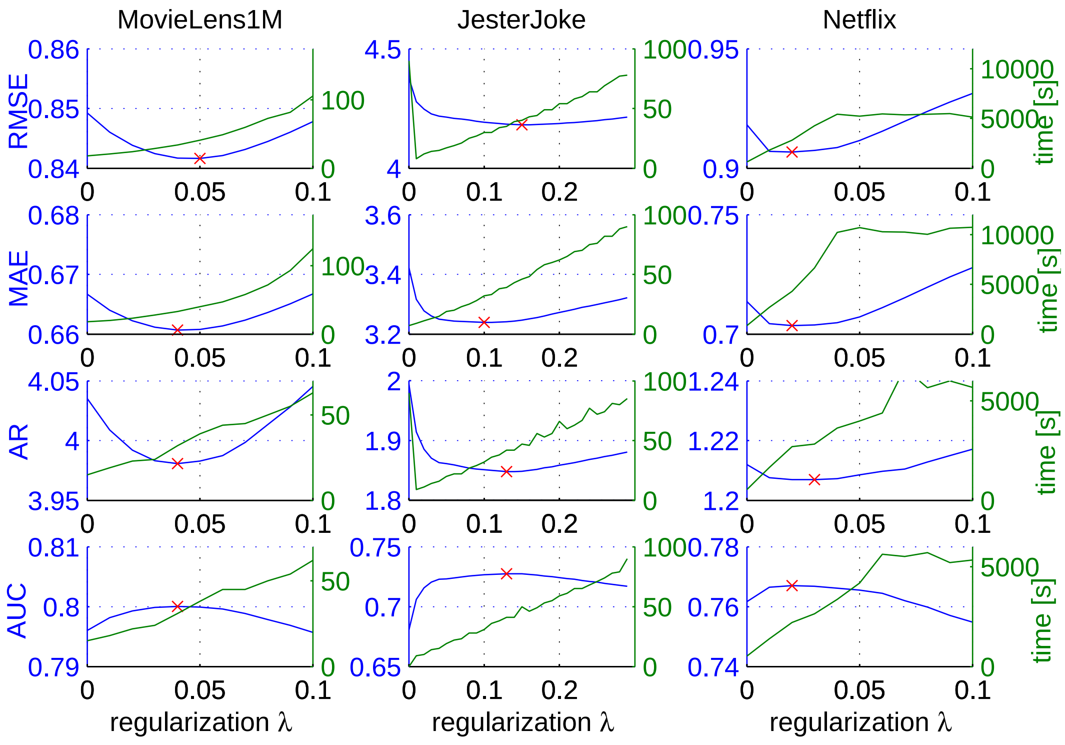

Figure 6.4: This figure visualizes the results of training a SVD++ model on different datasets and error measures and influence of the regularization.

6.2.5 Conclusion

We empirically analyzed the influence of L2 regularization to training time and accuracy.

We found that for SVD, RankSVD and SVD++ a higher regularization clearly results in an increased training time. For the AFM the result is not that clear. When analyzing the AFM with respect to the RMSE or MAE on the MovieLens1M dataset we see a strong increase of training time with higher regularizations. By contrast, on the Netflix dataset, training time decreases for regularizations greater than 0.02.

The accuracy achieved depends on a proper regularization. Unfortunately, we can not make any statement on guidelines to find a proper regularization. The ideal regularization is different for every algorithm, dataset and error measure. The only thing we found to be true in general is that too high regularizations can harm more than too low ones. Hence, it seems that in the case of not having the time to carefully tune the regularization, it is better to use a lower regularization. For real world systems using a lower regularization has also the benefit of a shorter training time.

The empirical results within this section show the benefit of modelling the selection bias. The SVD, which is the basic latent factor model, has a RMSE of 0.9190. The AFM, which only models the selection bias, has an RMSE of 0.9497. By combining the SVD and the AFM idea into the SVD++ model, the RMSE on the Netflix data improves to 0.9068. This nicely shows the value of modelling the selection bias on the Netflix data. In contrast on the Jester Joke dataset the SVD++ model performs slightly worse. The SVD has an RMSE of 4.08 and the SVD++ has 4.18. This is a result of the fact that the Jester Joke dataset has no selection bias, so introducing additional features for it does not improve the results. Due to the additional parameters the results become even slightly worse.

Another very important result is that the best algorithm is the same regarding all four error measures. The exact error values can be found in the tables in Appendix A. The SVD++ is the best algorithm on the Netflix dataset in terms of RMSE, MAE, AR and AUC. On MovieLens1M and Jester Joke the standard SVD is slightly better on all four error measures. This means if someone is interested in the best algorithm on a specific dataset, it suffices to use one error measure. This is different to the result reported by Gunawardana and Shani [14], who report that the choice of the error measure plays a major role for finding the best algorithm for a given dataset. In our opinion the different results are based in the algorithms used. The factor models used by us give good results for accuracy and ranking error measures, whereas the item KNNs used by Gunawardana and Shani produce bad results in terms of accuracy metrics like the RMSE.

6.3 Feature Size of Factor Models

The feature size is a very important meta parameter of factor models. It controls the number of latent features used to represent a user or item. So it controls the overall model complexity. It is obvious to see that the training time and the prediction time grows when we increase the feature size. Also the memory consumption rises by increasing the feature size. So for real a world system this leads to the interesting problem of finding the right tradeoff between accuracy and space/time requirements.

In order to investigate the influence of increasing the feature size we try the SVD, AFM, RankSVD and SVD++ on the MovieLens1M, Jester Joke and Netflix datasets. For each algorithm dataset combination we use the optimal regularization for $N = 30$, found in Section 6.3. We increase the feature size from 10 to 100 using a stepsize of 10.

6.3.1 SVD

The SVD uses regularization of $\lambda = 0.03$ on the MovieLens1M dataset, $\lambda = 0.06$ on the Jester Joke dataset and $\lambda = 0.02$ on the Netflix dataset. The results are visualized in Figure 6.5.

The first thing to notice is that the training time increases linearly with the number of features used. The next interesting thing is that the training times are shorter for the AR and AUC error measure, which means that the minimum is reached after fewer epochs. On the accuracy side we observe, that more features lead to a higher accuracy for all dataset and error measure combinations. The improvement from 10 to 20 features is clearly the biggest, from 20 to 30 the improvement is already smaller and the improvement of more than 30 features is nearly invisible in the graphs.

6.3.2 AFM

For the experiments with the AFM we used a regularization of $\lambda = 0.02$ for the MovieLens1M dataset, $\lambda = 0.04$ for Jester Joke and $\lambda = 0.02$ for Netflix. The results are visualized in Figure 6.6.

We observe a nearly linear increase in training time with more features being used. The training seems to stop earlier if we optimize for the AR or AUC. On the Netflix dataset the AFM behaves as expected, additional features increase the accuracy. Only

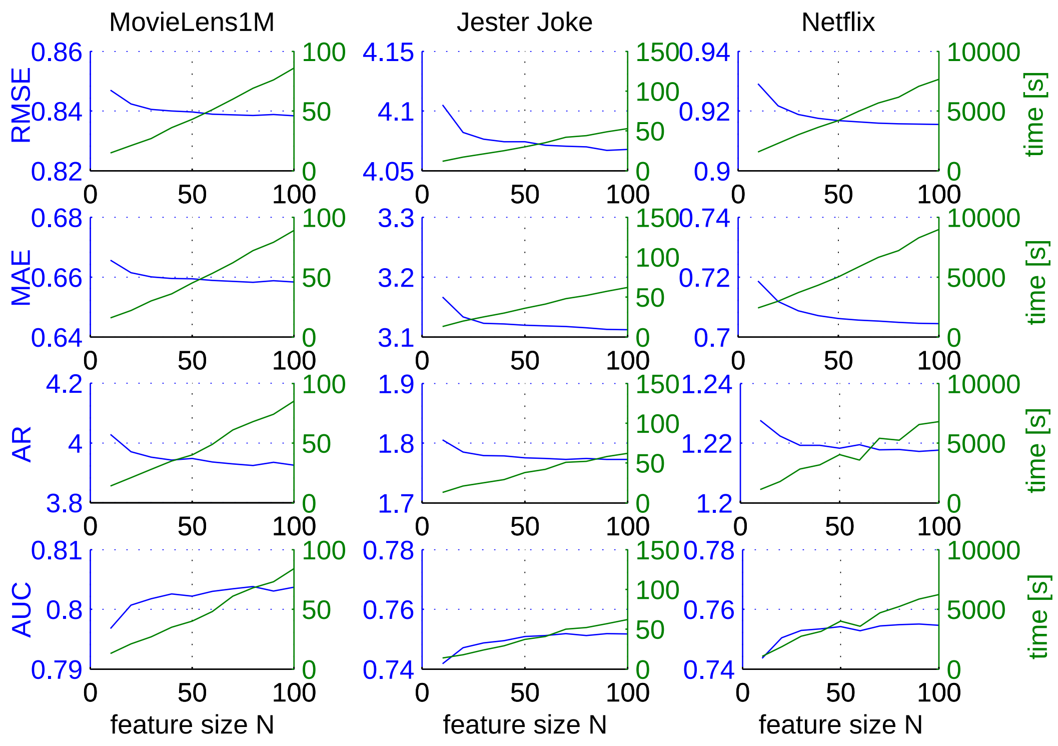

Figure 6.5: This figure shows the influence of the feature size N to the accuracy and training time of a SVD.

the MAE on the Netflix dataset gets slightly worse by using more than 50 features. On the Jester Joke dataset additional features also increase the accuracy, but the effect is very small. In our opinion this stems from the fact that the Jester Joke dataset does not have a selection bias in it, because the users do not select the jokes for voting by themselves. Moreover nearly 50% of the users have voted all jokes. So the AFM is not optimal for Jester Joke and additional features do not help. The behavior of the AFM on the MovieLens1M dataset is very strange, additional features decrease the accuracy on all four error measures under investigation. Initially we suspected this behavior stems from a too low regularization, but an increased regularization has not changed the behavior. This brings us to the conclusion that the AFM is not well suited for the MovieLens1M dataset.

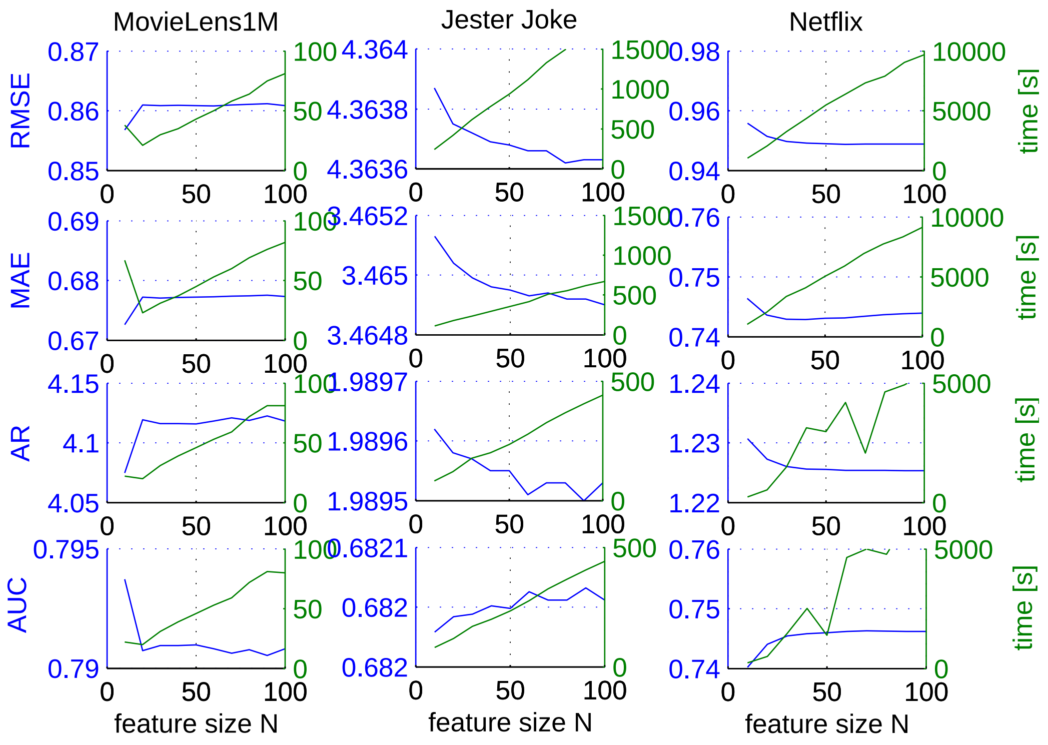

Figure 6.6: This figure visualizes the accuracy of an AFM with different feature sizes.

6.3.3 RankSVD

In the feature size experiments with RankSVD we used a regularization of $\lambda = 0$ for MovieLens1M and Jester Joke, and for the Netflix dataset we used $\lambda = 0.1$. The results are visualized in Figure 6.7.

Like in the other experiments with the feature size, the training time grows linearly with the number of features used. On the accuracy side we observe an interesting behavior. For the MovieLens1M dataset more than 50 features do not improve the accuracy, while we observe improvements for every added feature on the Jester Joke dataset. On the Netflix dataset more than 50 features seem to still improve the results, but only slightly.

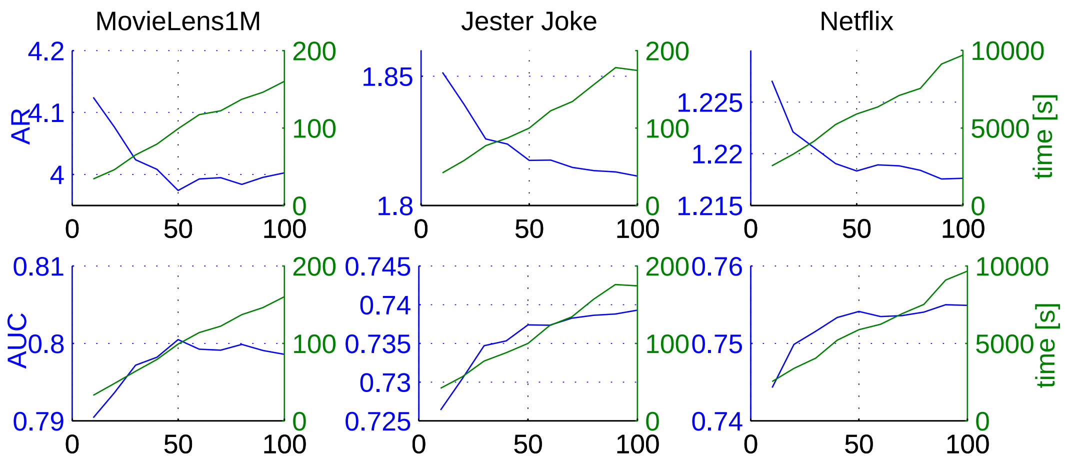

Figure 6.7: This figure shows the influence of feature size on RankSVD.

6.3.4 SVD++

The regularization parameters for this experiment are $\lambda = 0.04$ for the MovieLens1M dataset, $\lambda = 0.12$ for the Jester Joke dataset and $\lambda = 0.02$ for Netflix. The results can be seen in Figure 6.8.

For all datasets and error measures the training time rises linearly with the number of features used. On the Netflix data we observe that the training time for AR and AUC is much shorter, which simply means that the minimum is reached earlier. So the algorithm runs for fewer epochs. It is very interesting that this effect is not visible on the other datasets. On the accuracy side the behavior is as expected, every additional feature increases the accuracy.

6.3.5 Conclusion

We observed that the training time rised linearly with the number of features used. In the case one doubles the number of features the time and space requirements for one training epoch doubles exactly. This means that the observed linear increase of the complete training time is based on the longer runtime for one epoch, but the number of epochs stayed the same.

On the Netflix data we observed an interesting effect. On this dataset the AR and AUC tended to train fewer epochs, compared to RMSE and MAE.

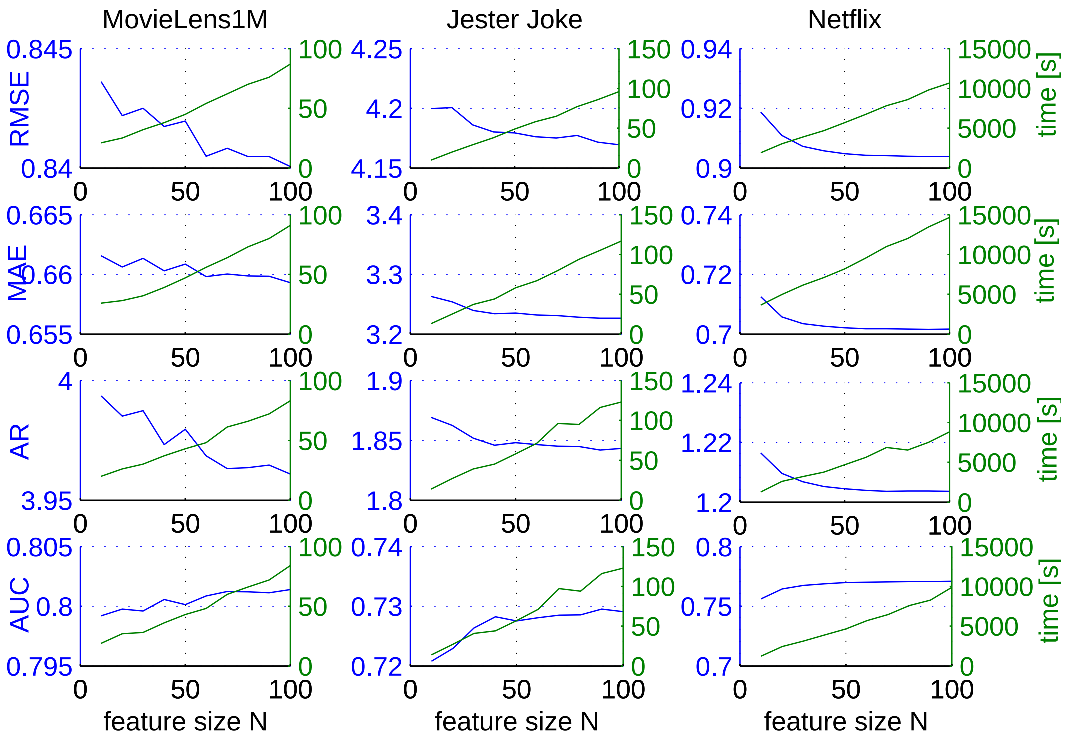

Figure 6.8: The figure shows the feature size experiment with a SVD++.

The accuracy of SVD, RankSVD and SVD++ increased with more features. The biggest improvements were always made by moving from 10 to 20 features. So for real world system one has to carefully think about how many features to use, because the training and prediction time rises linear, but the accuracy improvements for larger feature sizes are only marginal.

The accuracy of the AFM improves on the Netflix dataset only. On MovieLens1M and Jester Joke additional features do not increase the accuracy. The AFM is designed to model the selection bias, which is very strong in the Netflix data. In the MovieLens1M and the Jester Joke datasets it seems that there is little information in the fact that a user rated an item, hence, there is no selection bias. Therefore, the AFM, which only models the selection bias works badly on these datasets. Obviously, additional features do not help to increase the results of a AFM on MovieLens1M and Jester Joke.

6.4 The Relationship between Different Error Measures

Throughout this work we use different error measures for all our experiments. Our goal is to analyze the influence of the used error measure. In this section we are analyzing the relationships between the error measures directly. We investigate the relationships between RMSE, MAE, AUC and AR. By looking at the definitions of the error measures in Chapter 3 it is easy to see that a perfect algorithm has $E_{RMSE} = 0$, $E_{MAE} = 0$, $E_{AUC} = 1$ and $E_{AR} = 1$. A CF algorithm on a real world dataset will never predict user ratings without error, so the interesting question is how RMSE, MAE, AUC and AR are related on real world datasets.

The question of interest within this experiments is not to find a rule how to translate a RMSE into a AUC or something similar. In our opinion the main question of interest for real world systems is: Will we obtain totally different results by using a different error measure? All the error measures under investigation are very popular, so if someone trains a SVD and stops the training when the RMSE is minimal on a hold out set will the results be different compared to stopping the training based on the AUC. Does it make a difference if someone tunes KNN meta parameters to optimize the MAE; will the results look different to someones results who optimized everything to minimize the AR.

In order to investigate the relationship between the error measures, we recorded RMSE, MAE, AUC and AR on all training steps of the following models:

- Item KNN
- Item KNN V2
- SVD, N=30, $\lambda = 0.03$
- SVD, N=100, $\lambda = 0$
- SVD++, N=30, $\lambda = 0.02$
- AFM, N=30, $\lambda = 0.02$

The results on the MovieLens1M dataset are visualized in Figure 6.9, on Jester Joke in Figure 6.10 and on the Netflix dataset in Figure 6.11.

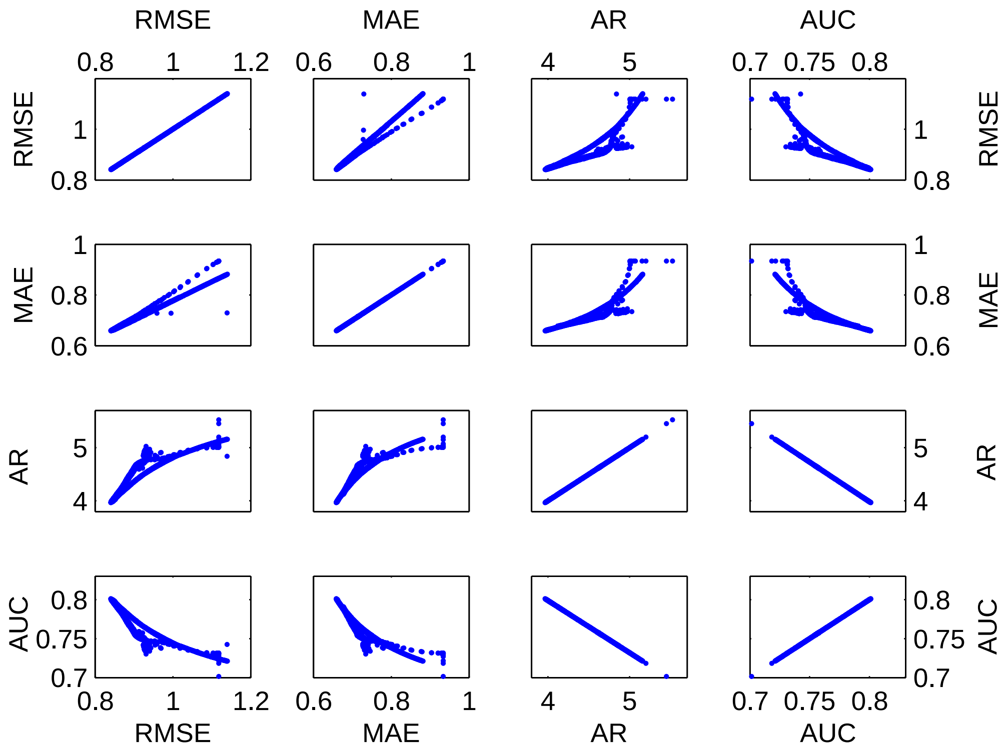

Figure 6.9: The Relationship between RMSE, MAE, AUC and AR on the MovieLens1M dataset.

By looking closely at the results on the MovieLens1M dataset in Figure 6.9, we observe that the AR and the AUC have a linear relationship. The relationship between the RMSE and the MAE is not exactly linear, but it comes very close. The relationship between the accuracy metrics RMSE and MAE and the ranking metrics AR and AUC is more complex and far away from linear. One very interesting thing to notice is that minimas of different error measures are closely connected. So the lowest RMSE values have a low MAE, AR and a high AUC.

In Figure 6.10 we plotted the results on the Jester Joke dataset. AR and AUC are again in linear dependency. The relationship between the other error measures seems to be more complex than the results on the MovieLens1M dataset suggested. On the MovieLens1M dataset the error values from the different algorithm runs and parametrizations had been fallen together into one graph. For the same experiment on the Jester Joke dataset the error values does not fall together into one graph. So it looks that we can not find a clear relationship between the error measures. But there

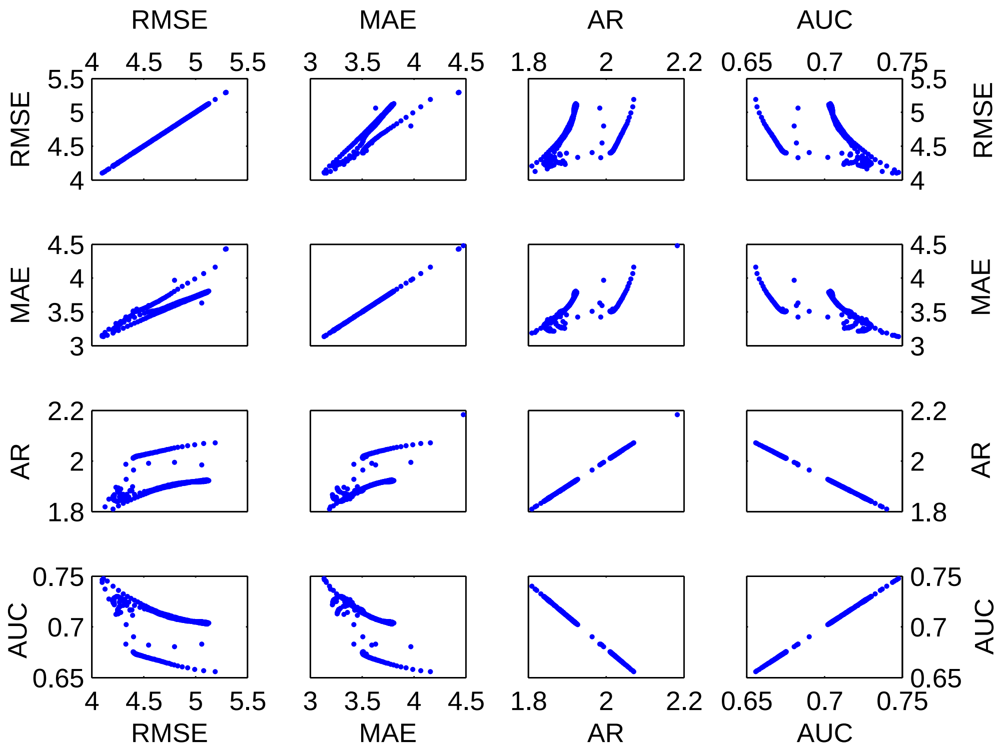

Figure 6.10: The Relationship between RMSE, MAE, AUC and AR on the Jester Joke dataset.

is one important thing to notice. We observed that for a fixed algorithm the minimas on all error measures are closely related. For example a SVD delivered the best RMSE after 135 epochs and the best AUC after 139.

In Figure 6.11 one can see the relationship between RMSE, MAE, AUC and AR on the Netflix dataset. In the cloud of points we can clearly see separated curves which stem from different algorithms. So we observe the same behavior as on the Jester Joke dataset.

6.4.1 Conclusion

The AR and AUC seem to be linearly related. So for this two ranking errors it makes no difference which one we use. So if one chooses to optimize for the AUC, the result will also be optimal in terms of the AR. The relationship between the accuracy measures RMSE and MAE is not so nicely linear, but these measures are still closely connected.

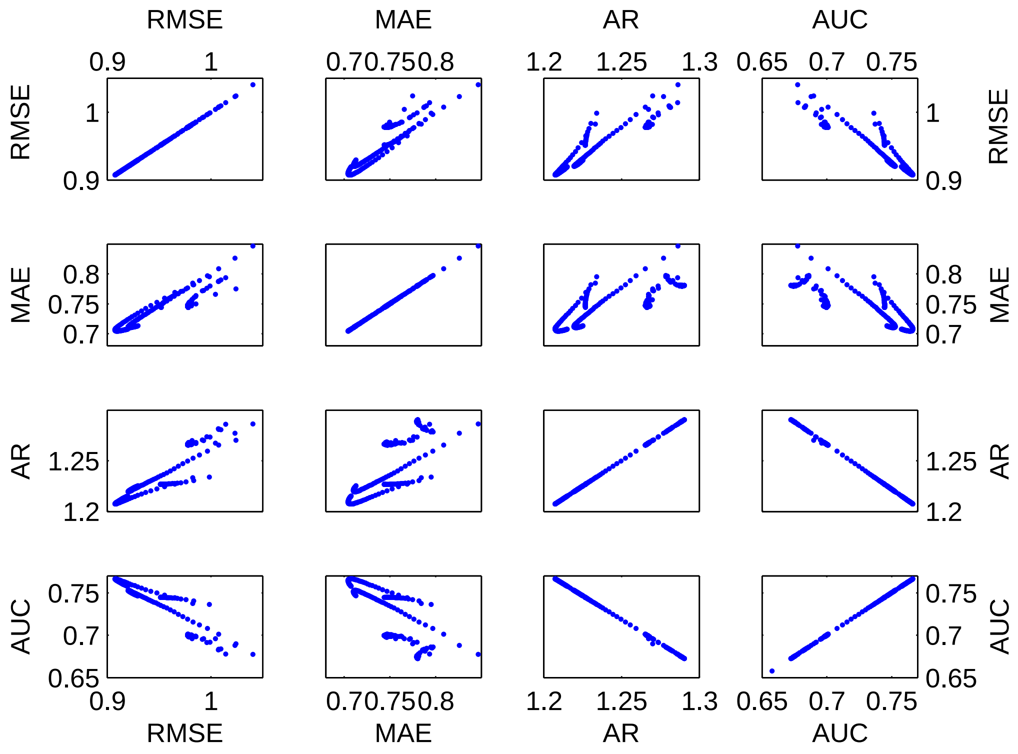

Figure 6.11: The Relationship between RMSE, MAE, AUC and AR on the Netflix dataset.

Between the accuracy metrics and the ranking metrics the relationship is very complex. In Figures 6.10 and 6.11 we observe that the curves which emerged in the point clouds do no fall together. Therefore it is not possible to calculate for example the AUC based on the RMSE. The curve describing the relationship between ranking and accuracy metrics is different for every algorithm dataset combination.

The most interesting fact we found in this empirical analysis of the relation between the error measures is that the optimas are closely related. For example, the number of epochs needed to reach the minimal RMSE is similar to the epochs needed to maximize the AUC. We observed this behavior for all algorithms, datasets and error measures under investigation within this work.

This means if one uses a SVD the resulting user and item features will be roughly the same regardless of the selected error measure. For example, let us train 2 SVDs one to minimize the RMSE and one for the AR. The first SVD trains as many epochs as needed to minimize the RMSE while the second runs for as many epochs as needed to minimize the AR. The number of epochs needed to reach the minimum RMSE will be roughly the same as those needed to reach the minimum of the AR. For this reason these two SVDs will generate very similar features for all users and items.

6.5 The Influence of the Fillrate

The Fillrate describes the percentage of known ratings. In real world datasets the rating matrix $\mathbf{R}$ is very sparsely filled, so the fillrate is small. It is well known that CF algorithms perform better with a higher fillrate. Or in other words, a CF algorithm needs information from the users and items in order to perform well. In order to investigate the change of the performance of the algorithms we used synthetic data and the Netflix dataset.

The synthetic dataset uses 5,000 users and 1,000 items. We investigate the performance of various algorithms with different parametrizations and evaluate the RMSE, MAE, AR and AUC. The number of ratings within the rating matrix $\mathbf{R}$ is changed in small steps from 5,000 to 5,000,000. This means that the average number of ratings per user goes from 1 up to 1,000. So in the beginning, where we have a low fillrate, the probe set $\mathcal{P}$ contains few users with more than one rating within the probeset, which is needed to calculate the AR and AUC. Thus for the lower fillrates these numbers are not meaningful. In Figure 6.12 one can see a comparison of various SVDs, and in Figure 6.13 we have a comparison between different algorithms.

Using the Netflix dataset we can do the same experiment as with the synthetic data. In order to generate smaller versions of the Netflix dataset with a lower fillrate, we simply generate random subsamples of the training data. The fixed Netflix probe set is not changed. Due to the subsampling it is possible that some users or items do not have ratings. In Figure 6.14 one can see the results on the Netflix dataset with number of training ratings ranging from 500,000 to 100,000,000.

6.5.1 Conclusion

All algorithms have improved the results with a higher fillrate. It is very interesting to see that the KNN type algorithms perform badly for too few ratings, which is an

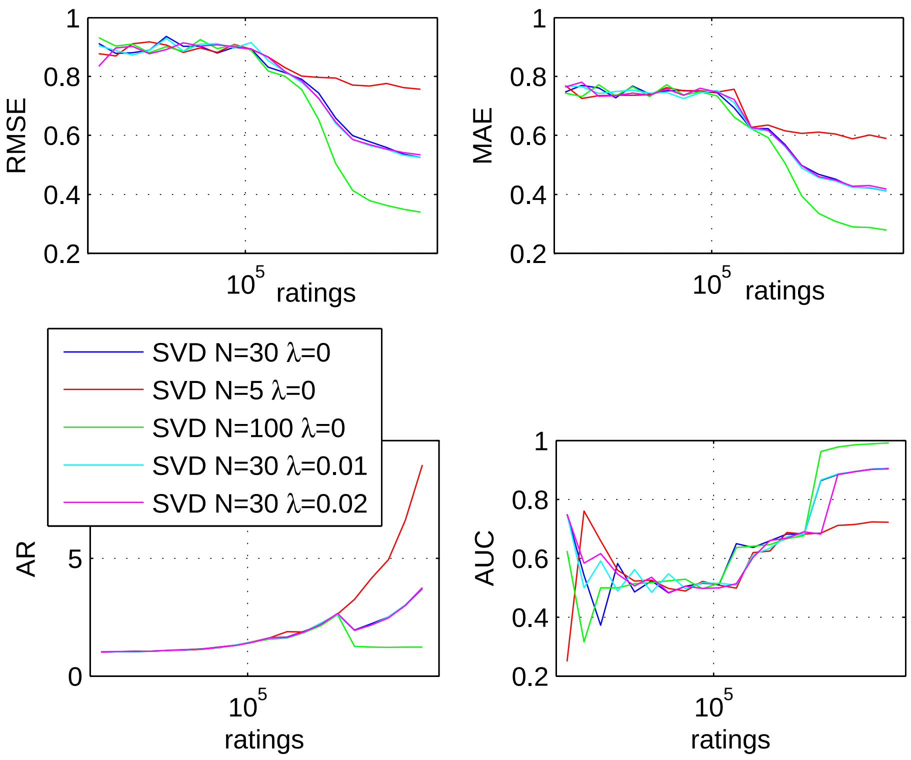

Figure 6.12: This figure shows the RMSE, MAE, AR and AUC of various SVDs on a synthetic dataset with a changing number of ratings. From 5,000 to 100,000 ratings within the matrix **R** there is no significant difference between the SVDs in terms of the RMSE and MAE. For more than 100,000 ratings SVD with $N = 5$ performs worst, while SVD with $N = 100$ performs best. The three SVDs with $N = 30$ and different regularizations perform nearly equivalent. So in this scenario regularization does not improve the results.

unexpected result. The performance of KNNs strongly depends on good preprocessing, and we have not used preprocessing within this work. Using preprocessing may have changed the results for KNNs. SVD and SVD++ models give good results with low and high fillrates. In order to perform really good on high fillrates the factor models need a lot of features in order to model complex user item relationships. The factor models with lots of features are doing as well as factor models with few features on a low fillrate, but doing much better with lots of ratings. This behavior suggests that factor models with lots of features are very good in terms of accuracy, regardless of the fillrate.

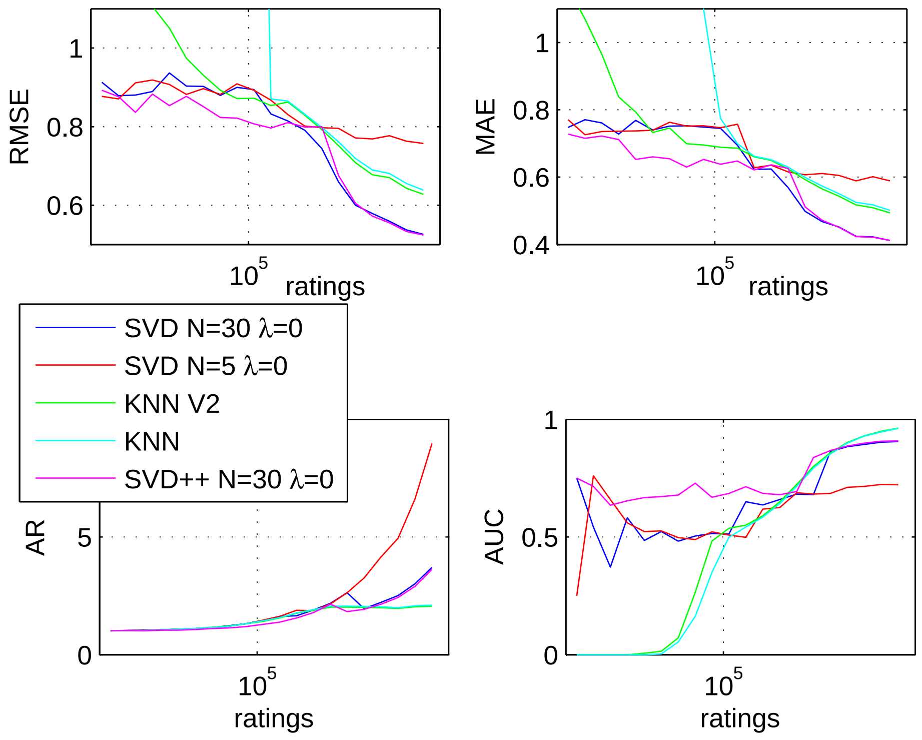

Figure 6.13: This figure compares the performance of SVDs and KNNs on a synthetic dataset with a changing number of ratings. From 5,000 to 100,000 ratings the simple KNN performs badly, the KNN V2 with the nonlinear correlation rescaling performs better. But already a simple SVD with $N = 5$ initially performs better. For more than 100,000 ratings the performance of KNNs situated between the SVD with $N = 5$ and the SVDs with $N = 30$. Within the synthetic data we have no selection bias, so the SVD++ does not perform better than a SVD.

This result is different to Cremonesi and Turrin [9] who compared the cold start behavior of factor models and item KNN on a IPTV dataset, and found item KNN outperform latent factor models. In our opinion the key difference is the way of training. We used a regularized stochastic gradient descent with early stopping, which introduces no problems during the cold start phase where the fillrates are very low.

We also compared SVD++ and AFM on the IPTV1 dataset from [9] and got significant better results. The results were obtained by exactly reproducing the way Cremonesi and Turrin simulated the cold start, namely to grow the dataset over time. Instead of using random subsampling to simulate the cold start, they used the time

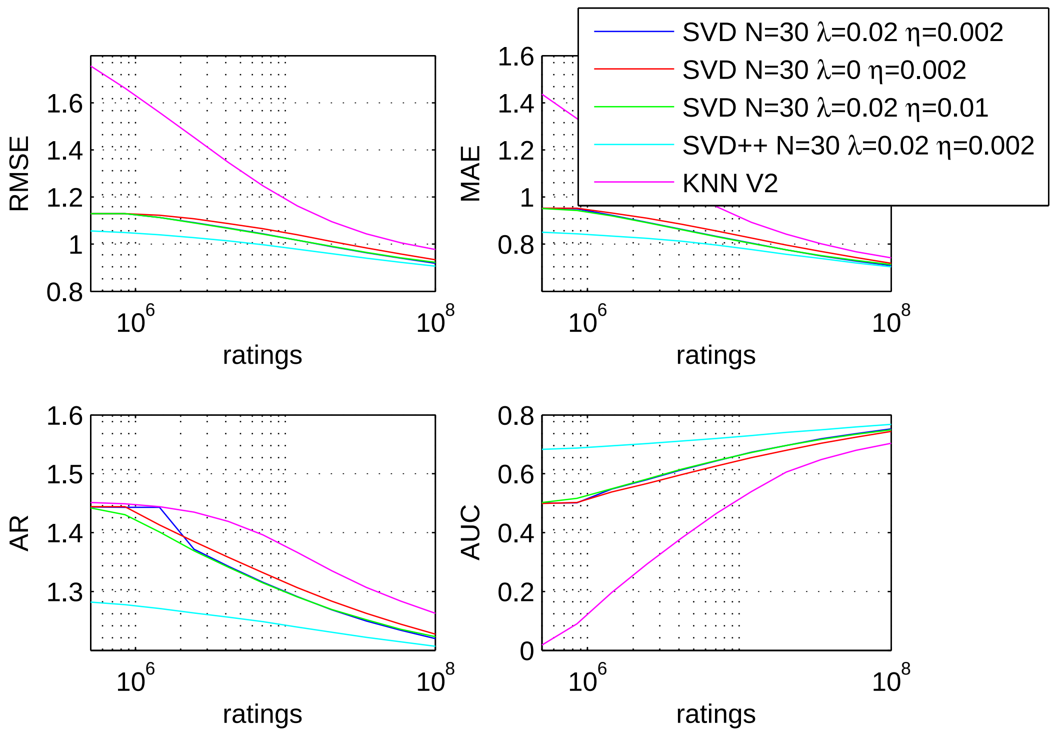

Figure 6.14: In this figure we visualize the results of various SVDs and a KNN V2 on the Netflix dataset. Through random subsampling we simulate datasets from 500,000 to 100,000,000 million ratings. The number of users and movies are unchanged. The SVD++ clearly performs best on all four error measures and from few to lots of ratings. The SVDs with different learn rates and regularizations perform slightly worse. In Section 6.2 we found, that $\lambda = 0.02$ is a good regularization for SVD models on the full Netflix dataset. In this figure one can observe that this regularization performs superior to $\lambda = 0.0$ from few to lots of ratings. The item KNN with nonlinear correlation rescaling (KNN V2) has big problems with too few ratings. For 500,000 ratings for 500,000 users results for 1 rating for the average user on the training set, and lots of the item/item correlations are undefined, which is not an ideal setup for a KNN algorithm. Preprocessing of the ratings can reduce the problem.

information from every datapoint.

6.6 The Influence of Observation Noise

A given rating of a user always includes a noise part. A rating is thought to consist of a real part which represents the real opinion of a user and a noise part. If a user rates an item several times the rating may change due to the observation noise. For the datasets used in this work, and for most available datasets, we cannot estimate the observation noise, because we only have maximal one rating per user/item pair. Anyway, our interest is not in estimating the amount of noise included in a given rating. We want analyze the impact of observation noise.

We are doing this experiment on synthetic data, with 5,000 users and 1,000 items. We use 2,000,000 ratings and add a Gaussian noise with a variance σ^2 ranging from 0 to 1. The results are visualized in Figure 6.15.

6.6.1 Conclusion

The amount of observation noise directly affects the accuracy level. A higher noise leads to higher errors. The interesting point is that all algorithms are effected in the same way. No algorithm reacted very sensitive to high noise levels.

For real world systems it is very important to design the user interface in a way that the observation noise is minimized, but the amount of expected observation noise does not influence the algorithm of choice.

6.7 Automatic Parameter Tuning - APT

Automatic tuning of meta parameters is very important. In Chapter 5 we described three direct search methods, the stochastic search, the coordinate search and the Nelder Mead algorithm. All of these methods work without gradients, can get stuck in local minima and are sensitive to the initial values. We use these methods for tuning meta parameters of collaborative filtering algorithms, so the main points of interest are the quality of the found parameters and the time needed to find those.

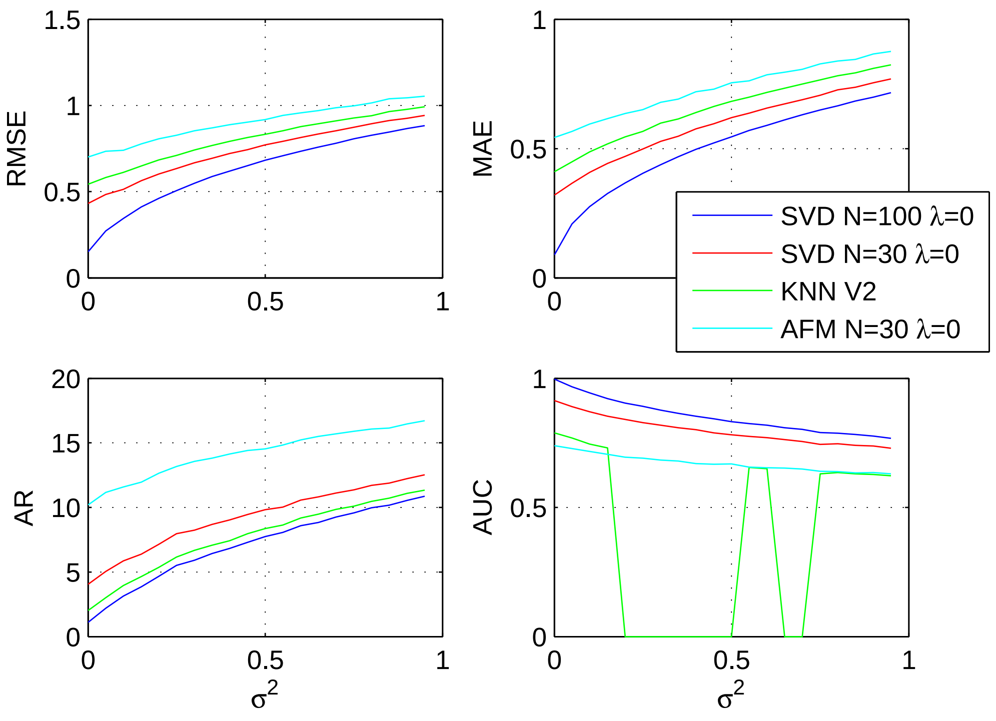

Figure 6.15: This figure shows the change in accuracy of SVD, KNN and AFM in terms of RMSE, MAE, AR and AUC for different levels of Gaussian observation noise on synthetic data. The graphs clearly show that higher variances σ^2 lead to lower accuracy. It is interesting to see that the accuracy of all algorithms is effected by the same amount. The jumps in the AUC of the KNN V2 stem from the parameter searcher which sometimes fails to correctly optimize the parameters for the AUC.

In order to compare the quality we made experiments on four datasets: Jester Joke, MovieLens1M, MovieLens10M and Netflix. To measure the error we used the RMSE, the MAE, the AR and the AUC.

The search time is also very important. By far the biggest part of the search time comes from the function evaluation. In order to get the error for a given set of meta parameters, one has to train the algorithm on the train set $\mathcal{T}$ and evaluate the performance on the probe set $\mathcal{P}$.

For the experiments we used the item KNN from Equation 4.15 which has four meta parameters to tune. The positive discrete parameter K to control the size of the neighborhood. The continuous parameter α to control the shrinkage of the item-item correlations and the continuous parameters δ and γ for the nonlinear rescaling. For the parameter tuning methods K is seen as continuous parameter which gets rounded to the nearest positive discrete value. So for the parameter tuning methods the error surface is locally flat dependent on parameter K. This results in a challenging task, which is well suited to compare the different search methods.

In Figure 6.16 one can see a plot where the KNN was optimized in order to minimize the RMSE. In the figure one can also see a method named “Nelder Mead Restart”, which initializes the simplex again when it got stuck in a local minimum. Figure 6.17 shows the KNN being optimized to minimize the MAE and in Figure 6.18 the AR is minimized. The results for maximizing the AUC are shown in Figure 6.19.

6.7.1 Conclusion

The results show, that the parameter searchers behave similarly on optimizing the RMSE and MAE. Also the results of AR and AUC are looking similar. For the accuracy metrics RMSE and MAE the differences in final accuracy are very small. In contrast the differences in final accuracy for AR and AUC are very big. So in our opinion the accuracy metrics produce a smoother error surface with fewer local minima, which is better suited for parameter optimization.

The Nelder Mead algorithm clearly achieves the fastest initial progress, but has problems in getting stuck early on. This is clearly visible in Figure 6.18 and 6.19. Reinitializing the initial simplex after getting stuck improves the final accuracy, but still this algorithm does not match the results of coordinate search.

Coordinate search delivered the best final results on all combinations of algorithms and error measures, but the initial progress is slower compared to Nelder Mead.

For getting a relatively good parametrization very fast, the combination of Nelder Mead Restart and an accuracy metric is very appealing. In the case that only the final accuracy counts or one wants to use AR or AUC then coordinate search is clearly the best choice.

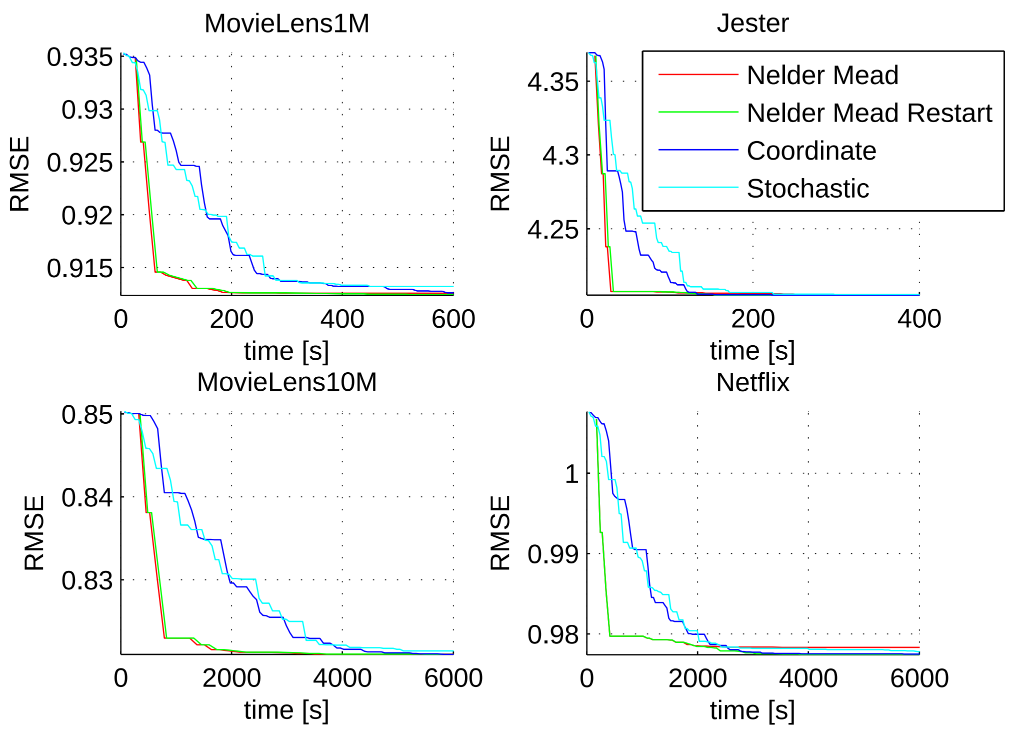

Figure 6.16: The figure shows the results of item KNN from Equation 4.15 being optimized to minimize the RMSE. On all four datasets the Nelder Mead algorithms achieve a lot of improvement after a very short time. Stochastic and coordinate search deliver very similar results, but have a slower progress compared to Nelder Mead. No algorithm has major problems in finding a good parametrization, and all methods find nearly equally good results. Looking closely on the bottom right of the graphs, one notices that coordinate search and Nelder Mead Restart found the best parametrization on all four datasets. Stochastic search has done worst on the MovieLens1M and MovieLens10M datasets, while standard Nelder Mead was worst on the Netflix dataset. Nelder Mead Restart has achieved the same final accuracy as coordinate and stochastic search on the Netflix dataset.

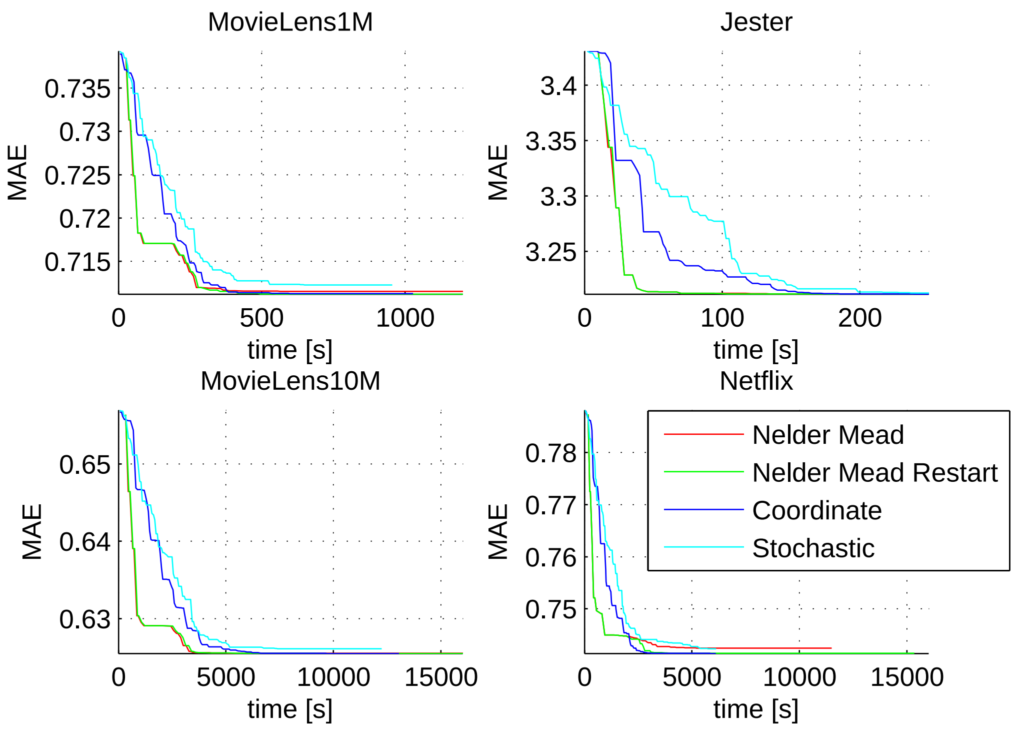

Figure 6.17: The figure shows the results of optimizing the parameters to minimize the MAE. On all four datasets Nelder Mead and Nelder Mead Restart has initially the fastest progress. The progress rate of coordinate and stochastic search is lower. The final accuracy of all methods on all datasets is very similar. Nelder Mead Restart and coordinate search are finding always the best parameters. The stochastic search delivers the worst results on MovieLens1M and MovieLens10M, and on the Netflix dataset standard Nelder Mead performs worst. So the results look very similar to those from minimizing the RMSE in Figure 6.16.

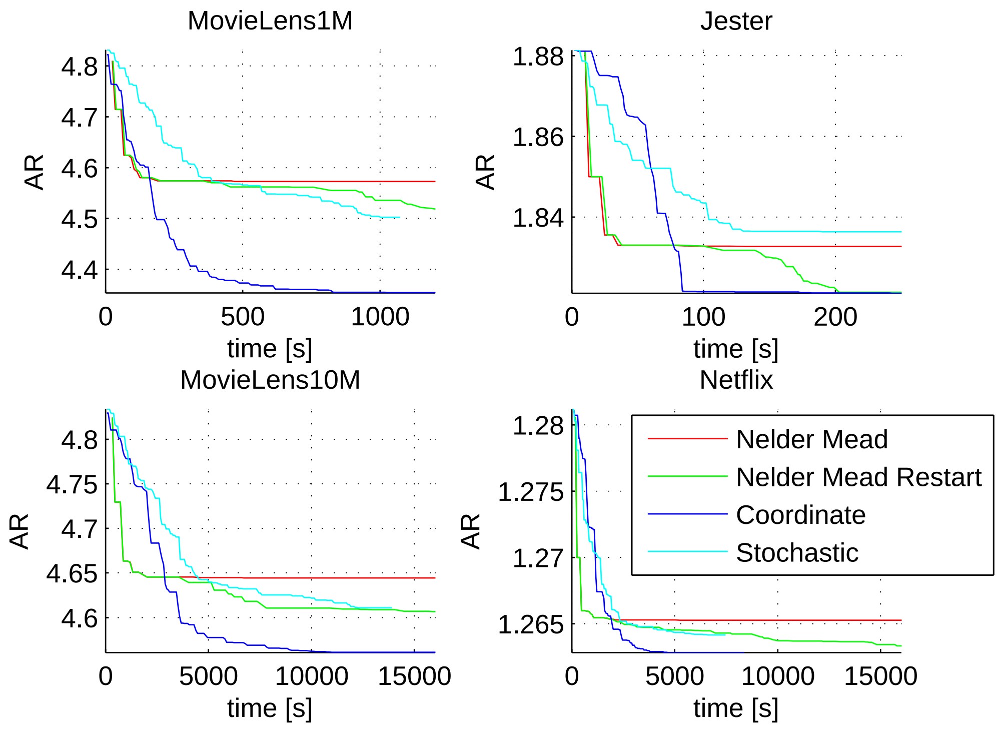

Figure 6.18: The figure shows the results of finding a parametrization to minimize the AR. In the first seconds Nelder Mead and Nelder Mead Restart are making the fastest progress on all datasets, but the difference to coordinate and stochastic search is not as big as for minimizing the RMSE and MAE. The final accuracy of the search methods varies strongly. This is a big difference compared to the results in Figure 6.16 and 6.17, where we optimized the RMSE and MAE. Optimizing the AR seems to be a much harder problem with a more complex error surface, so that the parameter searchers can get stuck in local minima more easily. The final accuracy of Nelder Mead was the worst in three out of four datasets. Restarting the Nelder Mead after getting stuck clearly improved the results. In the case of the Jester dataset Nelder Mead Restart is as good as coordinate search, but needs more time to achieve the same accuracy level. Coordinate search delivers the lowest AR on all 4 datasets, the results on the MovieLens1M and MovieLens10M are much better than those of the rest.

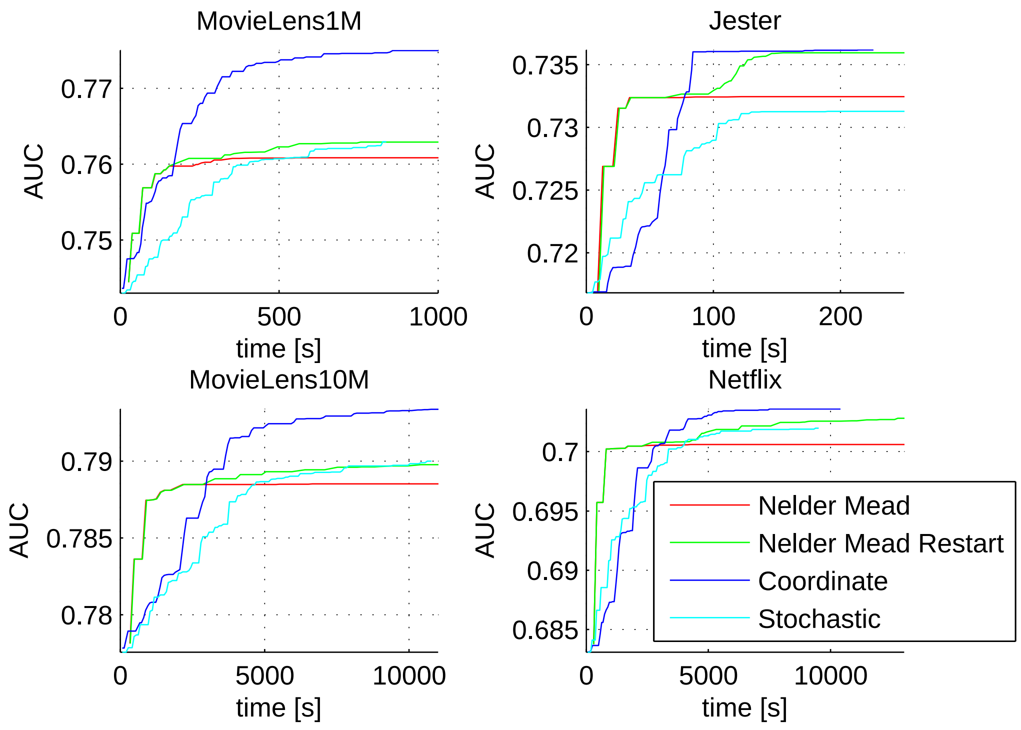

Figure 6.19: The figure shows the results of tuning the parameters of item KNN in order maximize the AUC. The results look like a flipped version of Figure 6.18, where the AR was minimized. Coordinate search performed clearly best on all four datasets. Nelder Mead has initially the fastest progress, but the worst final results in three out of four datasets. For minimizing the AR on the Jester dataset reinitializing the initial simplex after getting stuck improved the results of Nelder Mead to achieve the same accuracy level as coordinate search. The same is true for optimizing the AUC on Jester dataset. On the Netflix dataset the differences in terms of the final accuracy are the smallest and on the MovieLens1M the biggest.

Chapter 7

Conclusion

Contents

7.1 Conclusion

In this master thesis modern latent factor models like SVD, AFM, RankSVD and SVD++ were described and compared with neighborhood based approaches. All experiments conducted were performed on the MovieLens1M, MovieLens10M, Jester Joke and Netflix dataset or on synthetically generated data. Furthermore, we measured the accuracy in terms of the RMSE, MAE, AR and AUC.

For factor models we found that using a proper L2 regularization improved the results for every algorithm, dataset and error measure combination. The SVD, RankSVD and SVD++ showed the highest improvements. The problem is that the optimal regularization is different in every case. It turned out that in general it is better to use a too weak regularization than a too strong one. Additionally higher regularizations increased the training time. Thus, in the case of not having the time to carefully tune the regularization it is better to use a smaller or no regularization.

We found that the best algorithm for a given dataset is independent from the error measure used. This result stands in contrast to the recently reported result by Gunawardana and Shani [14], who report that the error measure plays a important role in finding the best algorithm. We speculate that the reason for the different empirical results is based in the selection of the algorithms.

Using more features always improves the accuracy and obviously leads to an increased training and prediction time. The improvements of the AFM on MovieLens1M and Jester Joke datasets are only marginal, which stems from the fact that the model is not suited for these datasets. The AFM models the selection bias which is not existing in these datasets.

In the conducted experiments we focused on different error measures and their relationship. We found that the minimas of the different error measures under investigation were closely related. For example, if a SVD is optimized to minimize the RMSE, the result will be similar to a SVD which optimizes the AUC. This means for the latent factor models the resulting features would be very similar.

We found that accuracy measures like RMSE and MAE are better suited for optimizing meta parameters with direct search methods, because of the smoother error surface. AR and AUC tend to have more problems with local minimas, so automatic parameter searchers can get stuck more easily. In our experiments structured coordinate search always found the best meta parameter setting. The Nelder Mead algorithm was found to converge much faster, but also to have more problems in getting stuck in local minimas.

We also investigated the influence of the fillrate of the rating matrix by subsampling the rating matrix. It was found that factor models always produced superior results compared to neighborhood based methods. This result stands in contrast to the results optained by Cremonesi and Turrin in [9], who found that item KNN outperforms a SVD in a simulated cold start. We speculate that latent factor models worked much better on sparse data in our setup than in [9] because we used regularized stochastic gradient descent with early stopping for training whereas [9] used standard SVD. We believe that standard SVD is less suited to these types of problems.

Bibliography

[1] Adomavicius, G. and Tuzhilin, A. (2005). Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. on Knowl. and Data Eng.*, 17(6):734–749.

[2] Balabanović, M. and Shoham, Y. (1997). Fab: content-based, collaborative recommendation. *Commun. ACM*, 40(3):66–72.

[3] Bell, R., Koren, Y., and Volinsky, C. (2007a). Modeling relationships at multiple scales to improve accuracy of large recommender systems. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 95–104, New York, NY, USA. ACM.

[4] Bell, R. M. and Koren, Y. (2007). Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *ICDM '07: Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, pages 43–52, Washington, DC, USA. IEEE Computer Society.

[5] Bell, R. M., Koren, Y., and Volinsky, C. (2007b). The BellKor solution to the Netflix prize. Technical report, AT&T Labs - Research.

[6] Breese, J., Heckerman, D., and Kadie, C. (1998). Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Proceedings of the Fourteenth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 43–52, San Francisco, CA. Morgan Kaufmann.

[7] Carenini, G. and Sharma, R. (2004). Exploring more realistic evaluation measures for collaborative filtering. *American Association for Artificial Intelligence.*

[8] Cortes, C. and Mohri, M. (2003). Auc optimization vs. error rate minimization. In *Advances in Neural Information Processing Systems*. MIT Press.

[9] Cremonesi, P. and Turrin, R. (2009). Analysis of cold-start recommendations in iptv systems. In *ACM RecSys 2009*, pages 233–236.

[10] Dasgupta, S., Johnson, C. R., and Baksho, A. M. (1990). Sign-sign LMS convergence with independent stochastic inputs. *IEEE Transactions on Information Theory*, 36(1):197–201.

[11] Funk, S. (2006). Netflix update: Try this at home. http://sifter.org/ simon/journal/20061211.html.

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

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

[14] Gunawardana, A. and Shani, G. (2009). A survey of accuracy evaluation metrics of recommendation tasks. *Journal of Machine Learning Research*, 10:2935 – 2962.

[15] Hanley, J. A. and McNeil, B. J. (1982). The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143:29–36.

[16] Herlocker, J. L., Konstan, J. A., Terveen, L. G., and Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems*, 22:5–53.

[17] Hirschberger, M., Qi, Y., and Steuer, R. (2007). Randomly generating portfolio-selection covariance matrices with specified distributional characteristics. *European Journal of Operational Research*, 177:1610–1625.

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

[19] Koren, Y. (2008). Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–434, New York, NY, USA. ACM.

[20] Lang, K. (1995). Newsweeder: Learning to filter netnews. In *Proceedings of the 12th International Conference on Machine Learning.*

[21] LeCun, Y., Bottou, L., Orr, G., and Muller, K. (1998). Efficient backprop. In Orr, G. and K., M., editors, *Neural Networks: Tricks of the trade*. Springer.

[22] Nathanson, T., Bitton, E., and Goldberg, K. (2007). Eigentaste 5.0: Constant-time adaptability in a recommender system using item clustering. In *ACM Conference on Recommender Systems.*

[23] Nelder, J. A. and Mead, R. (1965). A simplex method for function minimization. *The Computer Journal*, 7(4):308–313.

[24] Paterek, A. (2007). Improving regularized singular value decomposition for collaborative filtering. *Proceedings of KDD Cup and Workshop.*

[25] Pessiot, J.-F., Truong, T.-V., Usunier, N., Amini, M.-R., and Gallinari, P. (2007). Learning to rank for collaborative filtering. In *9th International Conference on Enterprise Information Systems.*

[26] Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., and Riedl, J. (1994). Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, New York, NY, USA. ACM.

[27] Resnick, P. and Varian, H. R. (1997). Recommender systems. *Commun. ACM*, 40(3):56–58.

[28] Salakhutdinov, R., Mnih, A., and Hinton, G. E. (2007). Restricted boltzmann machines for collaborative filtering. In *ICML*, pages 791–798.

[29] Schafer, J. B., Frankowski, D., Herlocker, J., and Sen, S. (2007). Collaborative filtering recommender systems. In Brusilovsky, P., Kobsa, A., and Nejdl, W., editors, *The Adaptive Web*, pages 291–324. Springer-Verlag Berlin.

[30] Singer, a. and Nelder, J. (2009). Nelder-mead algorithm. *Scholarpedia*, 4(7):2928.

[31] Takács, G., Pilászy, I., Németh, B., and Tikk, D. (2007). On the gravity recommendation system. In *KDD Cup Workshop at SIGKDD 07*, pages 22–30.

[32] Töscher, A. and Jahrer, M. (2008). The bigchaos solution to the netflix prize 2008. Technical report, commendo research & consulting.

[33] Töscher, A., Jahrer, M., and Bell, R. (2009). The bigchaos solution to the netflix grand prize. Technical report, commendo research & consulting.

[34] Töscher, A., Jahrer, M., and Legenstein, R. (2008). Improved neighborhood-based algorithms for large-scale recommender systems. In *KDD Workshop at SIGKDD 08.*

[35] Wu, M. (2007). Collaborative filtering via ensembles of matrix factorizations. *Proceedings of KDD Cup and Workshop.*

Appendix A

Error Tables

The following error values stem from Section 6.2. All models were trained with a feature size $N = 30$. The regularization used is the optimal regularization, which was found individually for every algorithm, dataset and error measure combination. The reported error values correspond to the red crosses in the Figures 6.1, 6.2, 6.3 and 6.4. For the RankSVD it is not possible to predict ratings, which means that RMSE and MAE can not be computed. Therefore, the corresponding cells in the tables are empty. The best error values are written in bold face.

A.1 MovieLens1M

	E_{RMSE}	E_{MAE}	E_{AR}	E_{AUC}
SVD	**0.8402**	**0.6593**	**3.9467**	**0.8023**
RankSVD	x	x	3.9741	0.8005
AFM	0.8608	0.6772	4.1166	0.7909
SVD++	0.8416	0.6607	3.9808	0.8000

A.2 Jester Joke

	E_{RMSE}	E_{MAE}	E_{AR}	E_{AUC}
SVD	**4.0793**	**3.1242**	**1.7786**	**0.7498**
RankSVD	x	x	1.8257	0.7347
AFM	4.3623	3.4615	1.9895	0.6821
SVD++	4.1822	3.2393	1.8481	0.7275

A.3 Netflix

	E_{RMSE}	E_{MAE}	E_{AR}	E_{AUC}
SVD	0.9190	0.7087	1.2195	0.7528
RankSVD	x	x	1.2206	0.7516
AFM	0.9497	0.7429	1.2260	0.7455
SVD++	**0.9068**	**0.7035**	**1.2069**	**0.7671**