LUND UNIVERSITY

Tuning of Learning Algorithms for Use in Automated Product Recommendations

JAKOB JÄDERBO

Lund 2014

Mathematical Statistics
Centre for Mathematical Sciences
Lund University

LUND UNIVERSITY

Abstract

Faculty of Engineering
Department of Mathematical Statistics

Tuning of Learning Algorithms for use in Product Recommendations

by Jakob Jäderbo

In this thesis, we study the problem of predicting users' media preferences based on their, as well as other users' historical rating data. The model used is that a user's rating for an item can be explained by a sum of bias terms and an inner product between two vectors in some multidimensional feature space that is specific to the product domain. This model is fitted using a stochastic descent type method, the stochastic diagonal Levenberg-Marquardt method. The method is studied with respect to its stability and how fast it adapts its parameter estimates and it is found that these characteristics can be accurately predicted for linear regression problems, but that the dynamics become much more complex when training the nonlinear features. The concept of regularization and the tuning of regularization parameters with the Nelder-Mead simplex method is also discussed, as well as some methods for making the Python implementation faster by using Cython.

Acknowledgements

There is a lot of people I should thank for helping me get through this thesis. First of all I would like to thank my supervisors Andreas Jakobsson and Sören Andersen for all our discussions, their feedback, and for suggesting this project to me. I must also thank my examiner Maria Sandsten, as well as my opponent Ingrid Johansson, for giving me lots of good feedback on my writing. Further, I would like to thank the department of mathematical statistics for providing me with all the other infrastructure that I have used while writing this thesis. A special mention goes to James Hakim for all the computer support I received during the early phases of my thesis. I would also like to thank all the good people I have been sharing office with for a few months now, it sure wouldn't have been the same without you. Last, but not least, I would like to direct an additional thanks to Annika for proofreading and, more importantly, motivating me to get through the last stages of my thesis.

Contents

Chapter 1

Introduction

1.1 Background

To find a good song to listen to today is no longer about browsing the local music store for new albums. Today we have access to millions of tracks over the internet for small monthly charges, or even at the cost of hearing the occasional advertisement. But having access to more than 20 million tracks on-demand means the problem is no longer whether the music a user may like is available, but whether the user is able to find it. This is an example of where the recommender systems steps in. Recommender systems and related technologies are found in many of the web services we use today. Not only in the context of music recommendation, but also when it comes to recommend other products in web stores, suggesting friends on social networks, deciding which news items that gets featured on the main page of a news site and displaying the advertisements that a given user may be most likely to be interested in.

The purpose of a recommendation system is to predict what content might be suitable for a user based on their previous behaviour such that it may be recommended to the user. For most applications this means ranking the items from most to least likely to suit a given user and presenting a selection of the top candidates to the user. From a user perspective, a good recommendation system may make the service easier and more enjoyable to use as it is more adapted to their preferences. For the service provider on the other hand a good recommender system can mean more loyal customers, more sales and better understanding of its customers.

1.2 State of the Art

Algorithms for recommender systems are typically divided into algorithms that base their recommendations on knowledge about the items they recommend, and algorithms that base their recommendations based on how users tend to interact with the recommender system. Using data on how users typically interact with the recommender is referred to as collaborative filtering and means that the recommender system learns to predict the users' behaviour based on how they and other users of the service have acted before. An example of collaborative filtering is the neighbourhood-based recommender systems that work by defining some form of similarity measure and use that to decide which users or items that are similar to each other. If users that are similar to you tend to like an item you haven't seen yet, then it is probably a good item to recommend to you. Neighbourhood-based recommenders are very common as they are easy to implement and still offer good performance. Neighbourhood-based recommenders can be either user-centric, giving recommendations based on what users similar to you have liked, or item-centric, recommending items that are deemed similar to the ones you like.

Another form of collaborative filtering is to approximate the incomplete matrix of ratings as a product between two low rank matrices such that the product explains the known ratings as good as possible. The method is related to the singular value decomposition [1] and infers a set of potential features in the data that can be used to predict the missing ratings. Matrix factorization methods were found to perform well in the Netflix challenge [2], where a million dollar price was offered to the team that could make a recommender that was better at predicting a user's preferences than the system developed by Netflix could, and have since been used extensively. Many recommender systems also use information about who liked what and how people are connected such that items can be recommended based on what the user's friends or people with similar demographics have liked.

Another approach is to use content based filtering to make recommendations. A prime example is the Music Genome Project [3] used by the music streaming service Pandora Internet Radio. Pandora employs experts to quantify the features of every song in their database which in turn are used to recommended items with similar features to the ones they liked in the past. Basic features such as tempo and pitch, can also be extracted directly from the audio files which may be used to give smooth transitions between songs in a generated play list.

There are also methods that does not fall into either category of recommendation methods. Such methods are typically hybrid methods that use information about both content and user interaction to make recommendations.

1.3 Scope of this Thesis

In this thesis, we study the problem of training a model for predicting ratings by capturing various biases in how the users' rate items, as well as the interaction between users and items. The methods studied for doing so are the stochastic gradient descent method and the stochastic diagonal Levenberg-Marquardt method; the main question concerned is how to best tune the methods for speed, accuracy and stability. The methods are implemented in Python and as the methods are computationally demanding some methods for reducing the memory demands and computation time are also discussed.

The thesis is restricted to computations using only interaction data; no metadata such as time stamps, demographic information and taxonomy, is being used. The evaluation is restricted to prediction accuracy measured by the root mean square error. Algorithms are limited to their single thread versions and are run in an offline setting.

1.4 Notation

In this thesis, scalars are denoted by plain letters e.g. a, A, α, vectors by lower case bold letters e.g. $\boldsymbol{a}, \boldsymbol{\alpha}$, and matrices by upper case bold letters e.g $\boldsymbol{A}$. Model parameters and optimizer hyperparameters related to a specific user have the subscript u while parameters related to a specific item have the subscript i. Hyperparameters related to a specific parameter have the parameter as subscript e.g. η_b is a scalar hyperparameter related to the parameter b.

1.5 Outline

Chapter 2 introduces two data sets, a small data set with movie ratings from Movielens and a larger data set with music ratings from Yahoo! Music. Chapter 3 discusses how to evaluate recommender systems and introduces the root mean square error on a validation set as an evaluation criterion. Chapter 4 discusses optimization methods for large scale optimization problems. In particular, the stochastic gradient descent method and the stochastic diagonal Levenberg-Marquardt method are introduced and analysed with regards to how different learning rates affect their performance and stability. In chapter 5, some basic linear regression models are evaluated for predicting ratings. Methods for avoiding overfitting are introduced in chapter 6. Chapter 7 explains how to use cross-validation to compare different models and how to use Nelder-Mead's simplex search to test many models in a structured way. Matrix factorization models for personalization are introduced in chapter 8. Some computational aspects are discussed in chapter 9 as well as methods for reducing the time and memory demands of programs written in a high level language by incorporating features from low level programming languages. Finally, chapter 10 contains teh conclusions that can be drawn by our results.

Chapter 2

Data Sets

2.1 Ratings Data Sets

In this thesis, we examined two data sets. Firstly, a small data set with 100,000 movie ratings, referred to as the Movielens 100k data set, collected by the GroupLens Research Project at the University of Minnesota. Secondly, a much larger data set consisting of 262,810,175 music ratings, referred to as the Yahoo! Music data set, collected by Yahoo! Both data sets are freely available on the internet [4, 5] and are well known within the research community.

2.1.1 The Movielens 100k Data

The Movielens 100k data is one of the oldest data sets in the studies of recommender systems. It consists of 100,000 ratings on a five star scale by 943 users on 1,682 movies, where each user has rated at least 20 movies. The data set is distributed freely for research purposes on the GroupLens web page as plain text files. The data files used in this thesis are the *ua.base* and *ua.test* files, which is a split of the data into a training set and a test set with exactly 10 ratings per user in the test set. In situations were a further validation set has been used in intermediate steps these have been taken by further splitting the training data. A sample of the data may look something like this,

```
3    353    1    889237122
3    354    3    889237004
3    355    3    889237247
4    11     4    892004520
4    210    3    892003374
```

where the first column is the user ID sorted in ascending order, the second column is the item ID, the third column is the given rating on a five star integer scale and the final column is a time stamp in seconds since 1970-01-01. There are also files containing further information about the movies such as title, genres and release year. For users there is information available about age, gender, occupation and zip code that may be used in modelling.

2.1.2 The Yahoo! Music Data

The Yahoo! Music data set is one of the largest data sets currently available to researchers. It consists of 262,810,175 ratings on a 100 point scale by 1,000,990 users on 624,961 items, where each user and each item has at least 20 ratings. The data set is publicly available through the Yahoo! Webscope program to researchers and academics. The data files used in this thesis are the ones created for the track 1 of the KDD Cup 2011 Challenge, which is split into training data, validation data and test data by taking the last six ratings by each user and adding those to the test set and taking the four ratings before those and adding those to the validation set. The total size of the training set is thus 252,800,275 ratings, the validation set is 4,003,960 ratings and the test set is 6,005,940 ratings [6]. For comparison a sample of the data looks something like this,

```
2|101
238557    90     4220    08:31:00
115200    100    4220    10:16:00
176889    100    4220    10:16:00
329058    100    4220    10:16:00
```

where the first row tells that this is the ratings made by user number two and that the next 101 rows are ratings made by the same user. For those rows the first column is the item id, the second is the rating, the third is a time stamp signifying which day the rating was made and the final column tells the time the rating was given. Additional text files contains information about which items were songs, artist, albums and genres and also how different items were related to each other. For privacy reasons there is however no more identifying information than that.

Chapter 3

Evaluating Recommender Systems

3.1 Offline Evaluation

The overall goal of a recommender system is to provide as accurate and satisfying recommendations as possible leading to increased customer satisfaction and revenue to the service provider. However, these goals are hard to measure in an offline setting as the users' true preferences will only be known to themselves. Had it been knowable, we would not need a recommender system at all.

Most metrics for offline evaluation rely on withholding a subset of the items rated by the users and see how well the recommendations compares to the test data. The reason for doing this, rather than using all available data for training, is that without these precautions there would be no way to tell how well the insights gained can be expected to generalize to previously unseen ratings. There have been a vast number of methods proposed for offline evaluation of recommender systems, see for example [7] for a treatment of the subject. Roughly, the methods can be divided into methods that view the recommendation problem as a prediction problem, methods that view the problem as a classification problem and methods that view it as a ranking problem. Seeing the problem as a prediction problem the recommender is evaluated based on prediction error over the test set. Common measures are the mean average error, MAE, and the root mean square error, RMSE. Methods based on viewing the recommendation problem as a classification problem instead evaluate how well the recommender can recognise a well liked item in the test set and tend to use some variation on precision and recall measures, where precision is a measure of how likely a recommended item is to be relevant and recall is a measure of how likely a relevant item is to be correctly classified. Ranking based approaches base the score on some function of the ranking errors of items in the test set. One problem all offline methods have in common is that they are all limited by only being able to measure how the user interacted with items that were found without the recommender. There is no way to tell if a recommendation of an item unknown to the user is correct or not based only on data of items known to the user. This means that it's hard to determine whether a serendipitous recommendation is correct or not in an offline setting.

3.2 Root Mean Square Error

For the purpose of this thesis the results will be evaluated by first training the predictor using the training data and then evaluating the prediction error as measured by the Root Mean Square Error (RMSE) on the test set using

$$RMSE(\boldsymbol{\theta}) = \sqrt{\frac{1}{m}\sum_{u}^{U}\sum_{i}^{I} M_{ui}(r_{ui} - \hat{r}_{ui}(\boldsymbol{\theta}))^2} \tag{3.1}$$

where m is the total number of ratings in the dataset, U is the number of users, I is the number of items, M_{ui} is a mask that is equal to one if user u rated item i and zero if the item was not rated by that user. Furthermore r_{ui} is the true rating given by user u for item i and $\hat{r}_{ui}(\boldsymbol{\theta})$ is the estimate of that rating given model parameters $\boldsymbol{\theta}$. The RMSE is mathematically elegant and simple to evaluate. The minimum is a least squares solution and there is a correspondence to making certain normality assumptions on the prediction errors. Comparing to the mean average error measure the impact of a large error is more severe under the root mean square error measure. Since the RMSE was used in the Netflix challenge it has become a standard in the recommender systems community, which means it is easier to find benchmark results using the RMSE than many other error measures. The RMSE is of the same dimension as the errors themselves, which makes it more intuitive to work with than the MSE or scalings of it, but the square root makes it harder to differentiate, which makes the optimization procedure more complicated. Another cost function with the same minimum is

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{u}^{U}\sum_{i}^{I} M_{ui}(r_{ui} - \hat{r}_{ui}(\boldsymbol{\theta}))^2, \tag{3.2}$$

which is a scaled MSE that can be used internally in the optimization procedure as it leads to simpler calculations.

Chapter 4

Optimization Methods

4.1 Batch Gradient Descent

Given a cost function, such as (3.1) or (3.2), the next step is to work out a way to minimize its value. The most basic such method is the batch gradient descent method, sometimes known as steepest descent. The method works by updating the parameter vector $\boldsymbol{\theta}$ at each iteration according to the update rule

$$\boldsymbol{\theta}(k+1) = \boldsymbol{\theta}(k) - \eta \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}, \tag{4.1}$$

where η is the step size or learning rate. The batch gradient descent is conceptually simple, but it suffers from slow convergence near the minimum. If line searches are used to select η, it approaches the minimum value in a zigzag pattern [8]. For the cost function (3.2) the derivative with respect to a single parameter θ is

$$\frac{\partial J}{\partial \theta} = -\frac{1}{m} \sum_{u}^{U} \sum_{i}^{I} M_{ui} \left(r_{ui} - \hat{r}_{ui}(\boldsymbol{\theta}) \right) \frac{\partial \hat{r}_{ui}(\boldsymbol{\theta})}{\partial \theta}. \tag{4.2}$$

4.2 Stochastic Gradient Descent

A problem with the batch gradient descent, and other optimization methods that calculate the derivatives (4.2) of the cost function (3.2) at each step, is that the computation of the gradient requires a summation over all the training examples. For a data set as large as the Yahoo! Music data set this means a summation of $m = 252,800,275$ training examples to take a single step. To increase the execution speed, one can instead use only a subset of the training data for each step. By averaging over a mini-batch or even taking just a single training example and updating according to that sub-gradient the algorithm can take a large amount of steps in the same time as the batch version could take only one. A sub-derivative of (3.2), being the derivative with respect to only one term of the sum, is

$$\frac{\partial J_{ui}}{\partial \theta} = -\frac{1}{m}\left(r_{ui} - \hat{r}_{ui}(\boldsymbol{\theta})\right)\frac{\partial \hat{r}_{ui}(\boldsymbol{\theta})}{\partial \theta}. \tag{4.3}$$

When the training examples are evaluated one at a time in, preferably in a randomized order, the method is referred to as the Stochastic Gradient Descent method (SGD). The idea behind stochastic gradient descent is that the average behaviour will cancel out most of the noise introduced by only considering a small sample at each step [9]. One of the reasons that we can achieve faster convergence using the stochastic gradient descent over the batch gradient descent is that it makes good use of redundancies in the data set. For a large data set with redundancies, one epoch of stochastic gradient descent, being a training cycle visiting all training examples once, may reduce the prediction errors more than a batch gradient descent would in several epochs [10]. To see why the stochastic gradient descent method utilises redundancies better, consider a data set where each training example appears twice. For a batch method the duplicates will only increase the time demands without speeding up the convergence. For a SGD method on the other hand each duplicate example will not matter, the algorithm still goes through just as many training examples from the same distribution per time unit.

```
Input: $S$: Set of training examples
Output: $\boldsymbol{\theta}$: Estimate of model parameters
Shuffle training examples
for $k$ in 1...MAX_EPOCHS do
    for rating in $S$ do
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\frac{\partial J_{ui}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$
    end
    if Stop Criterion met then
        Return $\boldsymbol{\theta}$
    end
end
Return $\boldsymbol{\theta}$
```

Algorithm 1: Stochastic Gradient Descent

Compared to batch optimization algorithms, stochastic descent algorithms may not converge as fast in terms of number of epochs towards the minimum of the empirical cost function – the cost function evaluated on the known training set. However, stochastic descent algorithms converge just as fast towards the minimum of the expected cost function [11]. The lower complexity also means that the algorithm can visit more training examples in less time, which is why they can still perform faster. SGD methods are popular as they are easy to implement and work well for very large data sets, however one drawback pointed out in [12] is that they require much manual tuning of hyperparameters such as the learning rates and convergence criteria.

4.3 Setting the Learning Rate

The noise introduced by visiting the training examples one by one when using the stochastic gradient descent method means that we will in general not converge to a specific point in the parameter space. Instead the parameter estimate will wander about in an area around a, possibly local, minimum [13]. The higher the learning rate, the faster the solution approaches the minimum initially. But a higher learning rate also means that the area in which the solution wanders about is larger. By selecting a suitable scheme for decreasing learning rates one can make the algorithm converge almost surely [9]. However, reducing the noise introduced by a high learning rate may not be as critical as one may think as overtraining may occur long before the noise regime is even reached [10].

Another issue to consider is how much of the previous parameter estimate that is retained and whether the update scheme is stable. The amount of the previous parameter estimate that is retained determines how fast the algorithm learns and how many training examples the current estimate is based on. A high learning rate corresponds to weighting the current training example high relative to the previous parameter estimate and as such the algorithm learns fast, but also forgets quickly. The SGD update step

$$\theta(k+1) = \theta(k) + \eta \frac{1}{m}(r_{ui} - \hat{r}_{ui}(\boldsymbol{\theta}))\frac{\partial \hat{r}_{ui}(\boldsymbol{\theta})}{\partial \theta} \tag{4.4}$$

can, given a model $\hat{r}_{ui}(\boldsymbol{\theta}) = a\theta$ or a sum of such terms, be rewritten as

$$\theta(k+1) = \theta(k)\left(1 - \frac{\eta a^2}{m}\right) + \frac{\eta a}{m} r_{ui} \tag{4.5}$$

which clearly shows how much of the previous parameter estimate that is retained and how much of the new training example that is taken into the model. For the most basic models, consisting of a sum of bias terms as will be discussed in chapter 5, the variable a in (4.5) is equal to one which further simplifies the form of the update. The dynamics of such an update scheme is mostly determined by the factor $\left(1 - \frac{\eta a^2}{m}\right)$. If the magnitude is larger than one the updates will tend to become unstable, and if the magnitude is lower the parameters will gradually adapt to new training examples while forgetting the old parameter estimates. The weight of a training example decrease as

$$\left(1 - \frac{\eta a^2}{m}\right)^j \approx e^{-\eta a^2 j/m} \tag{4.6}$$

after j updates. This knowledge can be used to make sure the learning rates are not too high or too low. If, for example, you have 10,000 training examples for learning the parameter θ but the learning rate is so high that the weight of all but 10 training examples is negligible then you are not taking advantage of your large data set and can probably reduce your learning rate. If on the other hand you have only 10 training examples in your data set but tune the learning rate to utilise the last 10,000 training examples your final parameter estimate will probably be heavily biased by the initial condition.

Setting the learning rate is not entirely a question about bias and variance, a very important aspect is to make sure that the updates are stable. A too high learning rate can lead to instability. Looking at how an update following (4.4) affects the error, $e_{ui}(k)$, at a specific training example we get

$$e_{ui}(k+1) \approx e_{ui}(k) \left(1 - \sum_{\theta \in \boldsymbol{\theta}} \frac{\eta_\theta}{m} \left(\frac{\partial \hat{r}_{ui}(\boldsymbol{\theta})}{\partial \theta} \right)^2 \right), \tag{4.7}$$

where the approximation lies in the assumption that

$$\frac{\partial^2 \hat{r}_{ui}(\boldsymbol{\theta})}{\partial \theta^2} = 0, \tag{4.8}$$

which will hold for all models studied in this thesis. The error dynamics in (4.7) sets an upper bound on the learning rates that will lead to stable updates. The stability bounds are constant for linear models, but may change over time for nonlinear models which will make these models much more complicated to tune. Further, we note that models with many model parameters put stricter bounds on stability than models with few parameters.

The information in (4.6) and (4.7) is enough to find a good learning rate for a specific parameter, but regression models used in recommender systems typically have a huge number of parameters, which may have a very different effect on the cost function (3.1). The data set is very unbalanced, meaning that some parameters may only be relevant to a few training examples, while others occur frequently. Using the same learning rate for all parameters leads to slow convergence in directions corresponding to parameters that occur infrequently in the training data. Setting the learning rate manually for all parameters would however not be a better option since there are typically too many parameters to tune individually. Luckily there are well structured ways of setting learning rates for many parameters at once. One such method is the stochastic diagonal Levenberg-Marquardt method.

4.4 Stochastic Diagonal Levenberg-Marquardt Method

The Stochastic Diagonal Levenberg-Marquardt (SDLM) method for selecting individual learning rates for parameters in stochastic gradient descent makes use of the local curvature to improve convergence. The learning rate η_θ for a specific parameter θ is chosen as

$$\eta_\theta = \frac{\eta_0}{\frac{\partial^2 J}{\partial \theta^2} + \gamma_1}, \tag{4.9}$$

where η_0 is the global learning rate and γ_1 is a parameter to prevent the learning rate from blowing up when the curvature is low. The inclusion of the second derivative of the cost function with respect to the parameter means that a lower learning rate is used for parameters that are encountered in many training examples. This ensures that the higher amount of training examples is used to reduce the variance of those parameters. Parameters that appear infrequently are given a higher learning rate such that the infrequent updates do not slow down the convergence. Using this approach means that we can set individual learning rates for millions of model parameters while only having to tune the two hyperparameters η_0 and γ_1. As a rule of thumb, the SDLM method converges about three times faster than a finely tuned ordinary SGD method [10].

4.4.1 Tuning the Stochastic Diagonal Levenberg-Marquardt Method

A parameter update using the stochastic diagonal Levenberg-Marquardt method has only two tunable hyperparameters. As such it is of great value to see how these hyperparameters affect the dynamics of the learning algorithm. Consider, as previously, a model with a linear parameter $\hat{r}_{ui}(\boldsymbol{\theta}) = a\theta$ or a sum of such parameters. For such a model the second derivative with respect to θ is

$$\frac{\partial^2 J}{\partial \theta^2} = \frac{a^2}{m}\sum_u^U \sum_i^I M_{ui} = \frac{a^2 N_\theta}{m} \tag{4.10}$$

and subsequently the learning rate η_θ of that parameter θ is (4.11)

$$\eta_\theta = \frac{\eta_0}{\frac{a^2 N_\theta}{m} + \gamma_1}. \tag{4.11}$$

The learning rates for linear parameters are thus scaled by the inverse frequency of how often they appear in the training data. This means that if the weight decline (4.6) of a parameter trained by the SDLM method is evaluated after one epoch's worth of training the weight decline will be the same for all parameters regardless of how many times they occur in the training data. Thus, assuming the stabilization hyperparameter γ_1 is small, all parameters converge at the same rate when using the SDLM method. The weight of an old parameter estimate declines by a factor of roughly

$$\left(1 - \eta_0 \frac{1}{N_\theta + \gamma_1 m}\right)^{N_\theta} \approx e^{-\eta_0} \tag{4.12}$$

per epoch, which can be used to set the number of epochs and the global learning rate η_0 to make sure all parameters are given appropriate learning rates that do not cause the parameter estimates to have either too much bias from the initial conditions or the estimates to be based on a too small subset of the available data.

To ensure stability we must require that all training examples result in a reduction of the error magnitude based on (4.7). For a nonlinear model this depends on the current parameter values, but for a linear model these bounds will only depend on the learning rates and can as such be estimated by considering what would be a worst case scenario for stability – typically an update involving several parameters with high learning rates, such as when both the user and the item involved have only one rating each in the training data. To make sure that even these extreme cases do not cause instability, the learning rate η_0 and stabilization term γ_1 in (4.9) must be set such that (4.7) remains stable, preferably with some margin to avoid oscillatory behaviour. A good way of thinking of the stabilization parameter is that the value $\gamma_1 m$ corresponds to a number of virtual ratings added for each parameter in the data set when the learning rates are calculated. This means that the parameters with few ratings are given lower learning rates as if they had been items with a larger amount of ratings. Since these virtual ratings are never encountered in training, the added stability of these parameters come at the cost of them having a higher bias from the initialisation than predicted by (4.12).

Based on these results we can set up a structured method for finding viable values of η_0 and γ_1 when using SDLM. First decide how many epochs that can be afforded based on how fast you can run your optimization algorithm. Then decide how much bias from the initial conditions that can be allowed to persist at the end of the optimization. Set η_0 as low as can be allowed using (4.12). If the updates are predicted to be stable using (4.7) you are done, if not – increase the value of γ_1 to make the updates stable.

4.5 Advanced Batch Methods

In comparison with batch gradient descent the stochastic gradient descent needs less time per update, but there are many batch methods that instead save time by converging in fewer iterations. To achieve this they use smarter methods to select search directions in order to avoid the zigzagging behaviour that plagues batch gradient descent methods. One such method is the Conjugate Gradient (CG) method which selects search directions using conjugacy which greatly reduces the number of times the same search direction has to be used. Other methods, such as the limited memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm [14], makes use of the local curvature of the cost function to find good search directions. The CG and L-BFGS methods require computation of the cost function's gradients on the full data set which can be an expensive operation on a large data set, but if mini-batches are used, it may not be too expensive. If multiple cores or GPUs are available the batch methods are much more suited to parallel computing than stochastic methods. The study in [12] shows that higher order methods are viable for large problems with close to one million parameters. They also found that, in a multi-core environment with GPUs the performance of CG and L-BFGS can be even faster than an SGD in the same environment. For the purpose of this thesis however the SGD method was chosen as it is easier to implement and offers good performance in single thread environments.

Chapter 5

Basic Bias Models

5.1 A constant predictor

The most basic predictor that can be considered is the constant predictor $\hat{r}_{ui}(\boldsymbol{\theta}) = \mu$. Minimizing the RMSE gives the following estimate

$$\mu^* = \arg\min_{\mu} \sqrt{\frac{1}{m}\sum_{u}^{N_u}\sum_{i}^{N_i} I_{ui}(r_{ui}-\mu)^2} \tag{5.1}$$

or equivalently minimizing the MSE (5.2), which is easier to differentiate,

$$\mu^* = \arg\min_{\mu} \frac{1}{2m}\sum_{u}^{U}\sum_{i}^{I} M_{ui}(r_{ui}-\mu)^2 \tag{5.2}$$

The μ that minimizes (5.2) is the mean rating. Using the mean rating of the training set to predict ratings in the test set results in an RMSE of 1.1238 using the Movielens data whether it's trained with SGD or by taking the mean value of the ratings. This is exactly the same results as reported by our Movielens benchmark [16], which isn't all that surprising since this model has no tuning to speak of.

Dataset	Model	RMSE	Benchmark
Movielens	Mean	1.1238	1.1238

For the Yahoo! Music data using the mean rating results in an RMSE of 38.2940. Our main benchmark results [15] for the Yahoo! Music data on the other hand reports an RMSE of 38.0617 using a constant predictor trained with stochastic gradient descent on the Yahoo! Music data set.

Dataset	Model	RMSE	Benchmark
Yahoo!	Mean	38.2940	38.0617

The question that arises is then: how can another estimate of the mean give a lower RMSE than the true mean? To understand that we need to study the parameter update equation. Taking a gradient descent step based on the cost function (5.2) above we update μ as

$$\mu(k+1) = \mu(k) + \frac{\eta_\mu}{m}(r_{ui} - \mu(k)), \tag{5.3}$$

where η is the learning rate. One notable feature is that at each update the estimator maintains a factor $\left(1 - \frac{\eta_\mu}{m}\right)$ of the previous update, which is the way the adaptive update can make use of information from several training examples. In chapter 4 it was explained how this adaptive update results in different weight being assigned to different training examples based on how recently they were encountered. As the Yahoo! Music data files are distributed sorted by user this means that an estimate made without shuffling will have a bias towards users with high user id. Depending on which learning rate that is chosen this takes the structure of the data set as a feature and trains the model on it and one can expect that another weighting leads to another estimate. If the learning rate and stop criterion are tuned to get the best result on some validation data, the resulting mean estimate may end up biased by the last training examples if these are more similar to the validation data than the average training example is. This effect is most prominent when the data sets have some form of structure as a structured data set may have many samples that are similar to each other appearing together. While this kind of structure may lead to better results than would have been possible for a shuffled data set, the results become dependant on an arbitrary ordering of users and may be hard to reproduce.

The validation and test sets have an equal amount of ratings for each user, while the training set has more ratings from the more active users. As the active users generally have different rating behaviour than the less active users, the mean of the training set's ratings may not be the same as the mean of the test set's ratings.

If we instead change the cost function on the training set such that each user is given the same weight when it comes to estimating the mean rating, rather than giving each rating the same weight, the mean estimate is increased from 48.8698 to 70.3431 on a rating scale of $0 - 100$. This much higher mean estimate reflects that a user that only rates a few items in general gives higher ratings than one that rates lots of items. A possible explanation could be that many users start out by only rating the items they actually like. Evaluating the prediction error using the weighted mean

$$\mu^* = \arg\min_{\mu} \frac{1}{2m} \sum_{u}^{U} \frac{1}{N_u} \sum_{i}^{I} M_{ui}(r_{ui} - \mu)^2 \tag{5.4}$$

results in an RMSE of 33.0353 on the test set which is a major improvement over using the true mean as an estimator.

5.2 A Basic Bias Model

The major improvement that could be made to the constant predictor by using a weighted mean was due to different users having different rating patterns. One way to improve the model is by adding individual bias terms b_u for each user to capture their personal user bias. It also makes sense that different items tend to get ratings reflecting their popularity or quality. This can be modelled by introducing item bias terms b_i. The constant predictor is thus replaced by a basic bias model (5.5).

$$\hat{r}_{ui}(\boldsymbol{\theta}) = \mu + b_u + b_i. \tag{5.5}$$

Note that this is a linear regression problem where the cost function can be written on the form

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \|\boldsymbol{B}\boldsymbol{\theta} - \boldsymbol{r}\|_2^2 \tag{5.6}$$

with $\boldsymbol{\theta}$ being a column vector $[\mu, b_{u1}, b_{u2}, \ldots, b_{iN_i}]^T$ containing all parameters, $\boldsymbol{r}$ being a column vector containing all ratings in the training set and $\boldsymbol{B}$ being a sparse matrix of ones and zeros selecting the bias terms relevant to each rating. A naive approach to estimate the parameter vector $\boldsymbol{\theta}$ would be to calculate the inverse or pseudo-inverse of $\boldsymbol{B}$ and use that to solve the linear equation system corresponding to (5.6). The problem is that the dimension of $\boldsymbol{\theta}$ is equal to $N_u + N_i + 1$, meaning that for the Yahoo! Music data set there are $1,625,952$ parameters and the $\boldsymbol{B}$ matrix is of dimension $1,625,952 \times 1,625,952$, making Gauss elimination infeasible from the sheer size of the problem. Attempting to store the pseudo-inverse of that matrix would be even more futile as it would in general be a dense matrix requiring several Terabyte of disk space to store.

If the point of using a stochastic gradient descent method was unclear when there was only a single regressor variable this is a completely different situation when it would be very hard to perform optimization without it. The bias terms all behave the same way and may be updated analogously to (5.3).

After setting aside a fraction of the ratings for validation the Movielens training data contained at least 9 ratings per user and at least one rating per item. This set an upper bound (4.7) on the global learning rate, $\eta_0 < 1.8$, if using (4.9) and the stabilization parameter γ_1 was close to zero. Experimentally it was verified that a global learning rate of 2.0 leads to increasing training error while a global learning rate of 1.7 resulted in the training process converging to an area close to a minimum. It was also found that an epoch could be performed in less than a second after some code optimization, see chapter 9. Knowing that training was somewhat cheap, 20 epochs of training with a global learning rate of 0.3 was used. Setting the global learning rate to 0.3 ensured that the initial condition $\boldsymbol{\theta_0} = \boldsymbol{0}$ would have small effect on the final results. These settings resulted in an RMSE of 0.9647 which is a clear improvement over the constant predictor and compares well with benchmarks of bias models using the same data set [16].

Dataset	Model	RMSE	Benchmark
Movielens	Mean	1.1238	1.1238
Movielens	Bias	0.9647	0.9656

It is possible to get slightly better results by using more epochs and a lower learning rate, but these improvements were found to be too small to be worth the additional effort as the model is expanded in later chapters.

The Yahoo! Music training set is almost $3,000$ times larger than the Movielens training set. The training set has at least 10 ratings per user and at least 4 ratings per item setting an upper bound on the global learning rate such that $\eta_0 < 5.7$ when (4.9) is used. The much larger size means that training was much more expensive and only a few epochs per hour could be afforded after code optimization. Prior to optimization the situation was even worse as a single epoch could take hours to perform.

The long computation time is the main reason why a good knowledge of how the learning rates affect the results is needed. Trial and error is expensive and trying to play it safe by setting a low learning rate and a high number of epochs can lead to simulations spanning days, which may not even give good results in the end.

Three epochs of training at a global learning rate of $\eta_0 = 2.0$ finished in slightly less than an hour with our Python implementation to be discussed later and resulted in an RMSE of 28.6588.

Dataset	Model	RMSE	Benchmark
Yahoo!	Mean	38.2940	38.0617
Yahoo!	Bias	28.6588	26.8561

Here the results differs even more from the benchmark results from [15], which may be related to the somewhat high learning rate and low number of epochs, resulting in a significant difference between the weighting of the first training examples of an epoch and the last. Experimentally, increasing the number of epochs and lowering the learning rates however seemed to have a small effect on the resulting RMSE. There are many other things that differ between the methods used, especially when it comes to the training procedure, but one thing that is performed in [15] that we have not discussed yet is regularization, which will be the topic of chapter 6.

Chapter 6

Regularization

6.1 Regularization

One problem with using a cost function like the RMSE defined in (3.1) when training is that there is no cost associated with model complexity. This means that there is a risk of overfitting the training data. When a model overfits the training data it may provide a good description of the training data, but fail to generalize to data that was not part of the training set. This behaviour comes from the model adapting to the noise specific to the training data rather than learning the underlying structure. Since even the most basic bias model with one parameter per user and item has thousands or even millions of degrees of freedom, depending on which data set we are studying, there is a very real reason to pay attention to the problem of overfitting.

A common way to counter overfitting is to modify the cost function to penalize the number or the magnitude of the model parameters. A regularized cost function with quadratic regularization costs, which penalizes the magnitude of the model parameters, is

$$J_{reg}(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{u}^{U}\sum_{i}^{I} M_{ui}(r_{ui} - \hat{r}_{ui}(\boldsymbol{\theta}))^2 + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2, \tag{6.1}$$

where λ is a non-negative regularization constant. For a linear model, such as the basic bias model (5.5), this leads to a cost function similar to ridge regression for which it has been shown that there exists positive regularization constants such that the resulting MSE of the minimized solution is lower than for the unregularized problem [17]. Regularization may also be motivated from a Bayesian framework by making assumptions on the distribution of model parameters, see for example [18]. For a quadratic cost function like (6.1) this corresponds to an assumption that the linear bias terms are normally distributed with zero mean and a variance proportional to $1/\lambda$ [19].

6.2 Regularized Stochastic Gradient Descent

If you calculate the gradient of the regularized cost function in (6.1), you get

$$\frac{\partial J_{reg}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{1}{m} \sum_{u}^{N_u} \sum_{i}^{N_i} \frac{\partial(r_{ui} - \hat{r}_{ui}(\boldsymbol{\theta}))}{\partial \boldsymbol{\theta}} (r_{ui} - \hat{r}_{ui}(\boldsymbol{\theta})) + \lambda \boldsymbol{\theta}, \tag{6.2}$$

which is straightforward to use in batch gradient descent. The regularization terms are not part of the sum from which the subgradients are taken which means some modifications must be made for stochastic gradient descent. Ideally only the elements of $\boldsymbol{\theta}$ that are accessed by the summation should be updated at each step to keep the updates computationally efficient. An update scheme that is true to the cost function in (6.1) and the gradient in (6.2) is

$$\theta(k+1) = \theta(k) + \eta_\theta \left(\frac{1}{m}(r_{ui} - \hat{r}_{ui}(\boldsymbol{\theta})) \frac{\partial \hat{r}_{ui}(\boldsymbol{\theta}))}{\partial \theta} - \frac{1}{N_\theta} \lambda \theta(k) \right) \tag{6.3}$$

which compensates the various update frequencies by scaling the updates such that the sum of all sub-gradients is the true gradient (6.2). The regularization hyperparameters, λ, are often different for different parameter types, e.g. one parameter λ for user biases, another for item biases and so on.

Many practitioners, such as [15], choose to regularize the update step rather than the cost function and end up with an update step such as

$$\theta(k+1) = \theta(k) + \eta_\theta \left(\frac{1}{m}(r_{ui} - \hat{r}_{ui}(\boldsymbol{\theta})) \frac{\partial \hat{r}_{ui}(\boldsymbol{\theta}))}{\partial \theta} - \lambda \theta(k) \right) \tag{6.4}$$

instead. This regularized update function does not compensate for different parameter frequencies which makes it is easier to maintain the stability of the updates as the scaling factor $\frac{1}{N_\theta}$ in (6.3), being the inverse number of ratings in the training set related to the parameter θ, may vary a lot in magnitude for unbalanced data sets. The drawback is that parameters that are updated often are regularized harder than parameters that are updated seldom. This corresponds to an assumption that the often encountered parameters are noisier, which is usually something you would not want to impose on your model.

6.3 Bias and Variance

The words bias and variance are often used when describing the trade-off between a very simple model that fails to fit the data properly and a complex model that fails to generalize to new data. Signs of underfitting is that the model performs badly on both the training data and the validation data and that adding more training data makes little difference in the results. If a model is suspected of underfitting the data it may be a good idea to increase model complexity by adding more parameters or reducing the regularization cost of the model parameters. If the model on the other hand is overfitting the data by having too many parameters and too little regularization it usually leads to a very good fit of the training data, but bad results on the validation data. Overfitting can be reduced by using more regularization, reducing the model complexity or by adding more training examples. The expected MSE when a model is applied to new data is a sum of squared bias, model variance and noise [18]. As methods that reduce variance tend to increase bias there is a trade-off between the variance and bias that has to made to attain the lowest possible MSE.

6.4 Selecting the Regularization Costs

Selecting the optimal regularization costs is a nontrivial problem. As neither the characteristics of the parameter distributions or the noise in the data is known it is hard to make an informed choice based on those. For a guess at the right order of magnitude one can look at how the regularization affects the steady state of the optimization algorithm when being repeatedly fed the same rating. By knowing how large the errors are when updates according to (6.3) results in no change to the parameter value we can get a feel for how much bias the regularization introduces. For a parameter $\hat{r}_{ui}(\boldsymbol{\theta}) = a\theta$ the steady state when receiving the rating r_{ui} is

$$\lim_{k \to \infty} \hat{r}_{ui}(\boldsymbol{\theta}(k)) = \frac{r_{ui}}{1 + \frac{m\lambda_\theta}{a^2 N_\theta}} \tag{6.5}$$

assuming that the update convention in (6.3) is used. If (6.4) is used the $\frac{\lambda_\theta}{N_\theta}$ is substituted by λ_θ to compensate for the absence of frequency rescaling. This means that for the update scheme (6.4) the steady state does not depend on how often a parameter is encountered. While this rule of thumb by no means tells us what the optimal regularization parameters are it can give a feel for whether a proposed level of regularization is to be considered large or small. For example, consider the regularization cost of a bias parameter for the Movielens data set. If $\lambda_\theta = 10^{-2}$ and the update convention (6.3) is used, that means a parameter that occurs 1,000 times in the training data will reach a steady state at barely 10 % of its steady state value without regularization, which seems like too strict regularization. Similarly, if the regularization hyperparameter $\lambda_\theta = 10^{-7}$ for the same data set that means that even a parameter that occurs only once in the data set will reach 99 % of its unregularized value at steady state, meaning that the regularization is not making much difference at all. This information can be used as a rough rule of thumb when it comes to setting the regularization parameters that can, for example, be used to find a good starting point for a hyperparameter optimization routine, such as the Nelder-Mead simplex search discussed in chapter 7.

6.5 Other Regularization Methods

Using L_2-regularization as in (6.2) is not the only way to reduce variance. There are other modifications of the cost function that add a penalty for models with high complexity. One such regularization method is the Akaike's information criterion, AIC, that adds a penalty on the number of parameters imposing sparsity on the model. Other well known methods that impose sparsity is the LASSO-estimator and the basis pursuit denoising which both modify the cost function by adding a penalty on the L_1-norm of the parameter vector.

Other methods take advantage of how the training process gradually reduces bias at the cost of increasing variance. At some point the variance will increase faster than the bias is reduced and the overall MSE will start increasing instead of decreasing. A subset of the training data can be used as a probe set to monitor when the MSE starts to increase. In [20] many methods that use this information to steer the training to more preferable regions are compared to traditional regularization methods for neural networks and are found to offer very good performance. The most basic such method is the early stopping method for which the training process simply stops when the MSE is found to be increasing on the probe set. This in effect means using the bias from the initial conditions as a regularizer. In fact a low learning rate or a low amount of epochs has a similar effect to ordinary regularization by stopping the parameter growth before they reach their unregularized steady state.

Other methods modify the learning rate or increase regularization as signs of overtraining starts to occur. The use of nonconvergent methods is based on the knowledge that gradient descent methods tend to converge slowly close to the minimum and that overtraining is likely to happen if training goes on for too long. As the objective of the learning algorithm is not to minimize the error on the training set, but to minimize the error on previously unseen data one can argue that convergence is not as important in a learning problem as it is in an ordinary optimization task. What matters is that the algorithm has learned as much from the training set as can be expected to generalize well to new data. Experiments with early stopping based on a 2% probe set did however not result in any better results than had already been achieved without it.

Chapter 7

Hyperparameter Optimization

7.1 Hyperparameter Over-fitting

When optimizing the algorithm to produce as good results as possible there are many options to evaluate. First there is the choice of which model to use, then there is the choice of optimization algorithm. If one chooses to use an SGD method one may have to decide whether it is better to shuffle the data to avoid imposing structure by the update scheme or if it is better to sort the ratings by their timestamps to use that structure to weigh the recent ratings higher. You also have to select which regularization method to use and how learning rates are chosen. If you decide to go with the SDLM method you have to set values for the hyperparameters η_0, γ and λ and so on. Combining the different approaches there are endless ways of configuring the optimization algorithm and model. To choose between many different hypotheses you need to be careful, as you may end up selecting an approach that outperformed the others purely by chance. This is another form of overfitting that is sometimes referred to as hyperparameter overfitting. Hyperparameter overfitting may occur if many different methods and hyperparameter settings are tested at once. A good method to reduce the risk of hyperparameter overfitting is to use cross validation.

7.2 Data Splitting and Cross Validation

As previously mentioned, the data sets are divided into three parts: training data, testing data and validation data. The training data, which is available during training, and the testing data, on which the final results are evaluated, are self-explanatory. The validation data set is used as an intermediary data set to avoid tuning the learning algorithm to the testing data. The procedure is to test all the algorithms and hyperparameter settings of interest by training them on the training data. The resulting models are then validated on the validation data to see which settings performed best. Once it has been settled which model gives the best results this model is evaluated on the test set to produce results that measure how well the model can be expected to generalize to previously unseen data.

It is important to maintain a good data hygiene and only take results on the validation data into account when deciding which set of parameters to evaluate on the testing data.

If testing data results have been used, consciously or unconsciously as a deciding factor for which method to use the results may be invalid due to fitting of the testing data. For this reason many machine learning competitions withhold testing data until all contest entries are submitted.

7.3 The Nelder-Mead Simplex Method

Having settled for a model to test and an optimization method there is the problem of setting the regularization cost λ, or more generally, different λ_θ for different parameter types. As the function evaluations, consisting of a full training session, are very expensive, a multidimensional grid search is not viable. A smarter method to search the hyperparameter space is the Nelder-Mead simplex method [21]. The Nelder-Mead Simplex method, see algorithm 2, is a derivative free optimization method for unconstrained problems that should not be confused with Dantzig's simplex method. It starts out with an initial guess around which it sets up a non-degenerate simplex, then it proceeds by evaluating the function value at the vertices of the simplex. Based on the function values, in this case the RMSE on the validation set after being trained with the proposed hyperparameters, the simplex is morphed according to a set of rules. First it attempts to reflect the vertex corresponding to the worst function value through the center of gravity of the rest of the simplex. If that seems to be a good search direction, it may attempt to expand the simplex to take larger steps in that direction. If, on the other hand, reflecting does not improve the function value, it may be that the optimal value is within the simplex, in which case the whole simplex is contracted and the process is repeated.

7.4 Optimizing Regularization costs for the Bias Models

Fixing the global learning rate η_0 in (4.9) to 0.3 and the number of epochs to 20, as in section 5.2, the Nelder-Mead method was used to find good values of the regularization parameters λ_{bu} and λ_{bi} for the Movielens data. The starting point was chosen as $\lambda_{bu} = \lambda_{bi} = 10^{-5}$. As the hyperparameters may span over several orders of magnitude the search was conducted on a logarithmic grid as proposed by [15]. This procedure resulted in suggested hyperparameters $\lambda_{bu} = 4.03 \cdot 10^{-5}$ and $\lambda_{bi} = 1.31 \cdot 10^{-5}$ and resulted in an RMSE on the test set of 0.9615 which is slightly better than without regularization.

Dataset	Model	RMSE	Benchmark
Movielens	Bias (unregularized)	0.9647	-
Movielens	Bias (regularized)	0.9615	0.9656

Changing the initialisation points for the Nelder-Mead solver either ended up with similar hyperparameter settings or got stuck in the somewhat flat area where regularization is too low to differentiate the solution from the unregularized model.

Performing the same procedure for the Yahoo! Music data with a learning rate of 2.0 and three epochs of training results in an RMSE of 28.1886 when the regularization costs are $\lambda_{bu} = 6.67 \cdot 10^{-9}$ and $\lambda_{bi} = 7.43 \cdot 10^{-11}$ respectively.

Dataset	Model	RMSE	Benchmark
Yahoo!	Bias (unregularized)	28.6588	-
Yahoo!	Bias (regularized)	28.1886	26.8561

The small difference between the regularized bias models and the unregularized bias models suggests that overfitting was not a major concern for the basic bias models considered so far. It may even be so that the models are underfitting the data and that a more complex model is needed to properly capture the structure in the data.

```
Input: $x_0$: initial point in $R^n$, $f$: function of $n$ variables
Output: $x_{opt}$: local optimum of $f$
Set up initial non-degenerate simplex $S$ around $x_0$
while 1 do
    $x_m \leftarrow \arg\min_S f(x)$
    $x_M \leftarrow \arg\max_S f(x)$
    if $f(x_M) - f(x_m) < tol$ then
        Return $x_m$
    end
    $x_g \leftarrow 0$
    for $x$ in $S$ not $x_M$ do
        $x_g \leftarrow x_g + \frac{x}{n}$
    end
    $x_r \leftarrow x_g + (x_g - x_M)$
    if $f(x_m) < f(x_r) < f(x_M)$ then
        $x_M \leftarrow x_r$
    else
        if $f(x_r) < f(x_m)$ then
            $x_e \leftarrow x_g + 2(x_r - x_g)$
            if $f(x_e) < f(x_r)$ then
                $x_M \leftarrow x_e$
            else
                $x_M \leftarrow x_r$
            end
        else
            $x_c \leftarrow x_g + 0.5(x_r - x_g)$
            if $f(x_c) < f(x_M)$ then
                $x_M \leftarrow x_c$
            else
                for $x$ in $S$ do
                    $x \leftarrow 0.5(x - x_m)$
                end
            end
        end
    end
end
Return $x_m$
```

Algorithm 2: The Nelder-Mead simplex method

Chapter 8

Personalization Model

8.1 Limitations of the Bias Model

The linear regression models used in past chapters have a serious limitation. That limitation is that every user would get the same ordering of the items with the only difference being that the ratings are shifted by their user bias. This means that every user would get the same recommended items. There is no notion of personalization in the models; the user bias and mean estimate only change the predicted ratings to reduce the RMSE, but does not result in any better recommendations. As the goal is to be able to provide relevant ratings to all users, rather than merely generating a list of the most popular tracks, the model needs to be improved to capture the user-item interaction.

8.2 Matrix Decomposition Models

One way to model the user-item interaction is to assume that each user can be associated with a number of preferences, and each item with a number of features. How well the item matches the users preferences is then some non-linear function of the user preferences and the items features. Matrix factorization models represent the user and item preferences as vectors in R^d and use the inner product

$$\hat{r}_{ui} = \boldsymbol{p}_u^T \boldsymbol{q}_i \tag{8.1}$$

between the user preference vector, $\boldsymbol{p}_u$, and the item feature vector, $\boldsymbol{q}_i$, to estimate how the user-item matching affects the users rating for the item. An interpretation is that, since $\boldsymbol{p}_u^T \boldsymbol{q}_i = \|\boldsymbol{p}_u\| \|\boldsymbol{q}_i\| \cos(\boldsymbol{p}_u, \boldsymbol{q}_i)$, the interaction can be understood by noting that users will be predicted to like the items the best when the angle between their preference vectors and the items feature vectors are small. Further, the magnitude of the feature vectors describe how much the preference matching contributes to the rating for that specific user or item. A user with a broad taste in music that uses a narrow interval of the rating scale would as such have a preference vector of low magnitude, while a user with a very specific music taste would be characterised by a preference vector of large magnitude. The favourite songs of the specific user would tend to have feature vectors pointing in a similar direction in feature space as the user's preference vector. In the article by [15], matrix factorization combined with the an extensive modelling of bias terms is used. If we take the bias modelling from chapter 5 and add inner product terms (8.1) we get a biased matrix factorization model for which the ratings are estimated as

$$\hat{r}_{ui} = \mu + b_u + b_i + \boldsymbol{p}_u^T \boldsymbol{q}_i. \tag{8.2}$$

8.3 Updating the Nonlinear Parameters

Consider the single feature model

$$\hat{r}_{ui} = p_u q_i \tag{8.3}$$

representing each rating as a product of two scalars. Differentiating the cost function (3.2) with respect to p_u using this very simple nonlinear model gives us

$$\frac{\partial J}{\partial p_u} = -\frac{1}{m} \sum_i^I M_{ui}(r_{ui} - \hat{r}_{ui}) \frac{\partial \hat{r}_{ui}}{\partial p_u} = -\frac{1}{m} \sum_i^I M_{ui}(q_i e_{ui}), \tag{8.4}$$

which differs from the linear case only in that the q_i in general are different for each item encountered in the sum. The summation over users not using the parameter p_u is also omitted. (8.4) gives us the SGD update rule

$$p_u(k+1) = p_u(k) + \eta_{p_u} \frac{1}{m} q_i(k)(r_{ui} - p_u(k) q_i(k)). \tag{8.5}$$

The important thing to note here is that the values of the item features will also be updated over time in an analogous fashion. This means that the learning will have nonlinear dynamics. One effect of the nonlinear dynamics is that it is much harder to set an upper bound on the learning rate with (4.7) as the stability when updating p_u depends on the values of q_i and the stability when updating q_i depends on the values of p_u.

The update (8.5) can be rewritten as

$$p_u(k+1) = \left(1 - \eta_{p_u} \frac{1}{m} q_i^2(k)\right) p_u(k) + \eta_{p_u} \frac{1}{m} q_i(k) r_{ui} \tag{8.6}$$

which highlights how the stability of the updates for p_u depends on the values of q_i it encounters. That is, if $q_i^2(k)$ is large the stability of the updates of p_u gets worse. For a given learning rate this constrains the parameters p_u and q_i to a rectangular area outside which the updates may be unstable. While the estimates are the same if $p_u = 1$, $q_i = 1$ or $p_u = 1/1000$, $q_i = 1000$ the latter will be more likely to cause instability. Regularizing the cost function to make sure solutions with smaller norms are preferred prevents one parameter from growing indefinitely while the other one shrinks.

8.4 Stochastic Diagonal Levenberg-Marquardt for Nonlinear Regression

The stochastic diagonal Levenberg-Marquardt method proved to offer a good way of setting the learning rates for linear regression with the basic bias model. It would be desirable to use the same approach for training the nonlinear matrix factorization model on the residuals from the bias model. The SDLM method uses the curvature of the cost function (4.9). Using the single feature model (8.3) as in (8.4) and taking the second derivative with respect to p_u we get

$$\frac{\partial^2 J}{\partial p_u^2} = \frac{1}{m}\sum_i^I M_{ui} q_i^2(k), \tag{8.7}$$

which is similar to (4.10) in form. There is however a very important difference between the linear case and the nonlinear case. The difference is that in the linear case the curvature, and, by (4.9), also the learning rate

$$\eta_{p_u} = \frac{\eta_0}{\frac{1}{m}\sum_i^I M_{ui} q_i^2(k) + \gamma_2} \tag{8.8}$$

is dependant on the current values of the parameters.

These updates makes sure that the preference vectors change more slowly for users that encounter many items with large feature vectors. However, this does not fully compensate for the factor $q_i^2(k)$ in (8.6) and the stability may be orders of magnitude worse than for the corresponding update of a linear model in terms of η_0. A problem that arises from the dependence between learning rates and parameters is that the learning rates have to be recomputed often. Evaluating the sum in (8.8) every time p_u is updated leads to a time complexity that we can not afford, especially so if the purpose was to save time and speed up calculations. The alternative of keeping all learning rates in memory and updating the ones that are affected at every update is not much better as it leads to both increased time complexity and increases memory demands by a large amount. It should however not be necessary to use the exact values of the learning rates all the time. In [10] it is suggested to use a running estimate of (8.7), in this case that would correspond to having a running average of the squared magnitude of the item preference values for items encountered during training for each user. This approach makes sure that memory and time complexity stays in the same order of magnitude, the drawback is that it comes at the cost of another hyperparameter to tune and more complex dynamics of the optimization algorithm.

One such adaptive scheme gives each user an additional set of hyperparameters for keeping track of the estimates of the mean square feature, $\hat{q}_u^2(k)$, for each dimension. The learning rate (8.8) can then be rewritten as

$$\eta_{p_u}(k) = \frac{\eta_0}{\frac{1}{m} N_u \hat{q}_u^2(k) + \gamma_2} \tag{8.9}$$

whereas $\hat{q}_u(k)^2$ is updated after each example as

$$\hat{q}_u^2(k+1) = (1-\alpha_q)\hat{q}_u^2(k) + \alpha_q q_i^2. \tag{8.10}$$

The adaption rate α_q can be set to e.g. $\frac{1}{N_u}$ analogous to a somewhat aggressive SDLM update that coincides with (8.8) in the case when the number of ratings made by the user, N_u, is equal to one. As $\hat{q}_u^2(k)$ in the denominator of (8.9) changes over time it is important to make sure γ_2 is nonzero as the denominator may otherwise become small and cause instability.

8.5 Stability and Learning Rate

As has been mentioned previously, there are situations when the stability of the nonlinear updates behaves differently than the stability of linear updates. (8.6) shows that an estimated user preference can become unstable if the corresponding item features are large, analogously estimated item features may become unstable if the estimated user preferences are large. The stability region is larger for lower learning rates, meaning that, unlike the linear case, the learning rate now affects the solutions that are attainable. How large the estimated values of the features will grow is hard to predict as the learning process is stochastic and there are multiple parameter settings that result in identical models. Regularization can make sure that more well-behaved models are preferred at convergence. However, due to the random initialization and the noisy training procedure there is still risk that the parameters may drift into unstable territory during training. If an adaptive learning rate scheme is used such as the SDLM method (8.9) the dynamics gets even more complex as the learning rates may span several orders of magnitude and will also exhibit nonlinear stochastic dynamics.

Experimentally it is found that the SDLM method does indeed go unstable easily. Using a single feature model $\hat{r}_{ui} = p_u q_i$ for the Movielens data it is found that parameters will blow up at a global learning rate η_0 as low as 0.03 when the stabilization parameter γ_2 is close to zero. That is a factor 60 lower than what would have been stable for a linear model on the same dataset. In fact, it is not that surprising the stability is bad when the stabilization parameter is low as the denominator of (8.9) may become close to zero. By increasing the stabilization hyperparameter γ_2, the stability is improved, but the allowable learning rates are more limited than for the linear model.

As the model is expanded to a more useful size with a 50-dimensional feature space the stability drops by even more as 100 parameters that may go unstable individually or in combination as they are updated together.

One might get the impression that the low threshold on the value of η_0 should mean that it takes lots of epochs to have any effect at all on the initial conditions. But in the previous analysis of forgetting factors the learning progressed at an even rate with the same weight ratio between the old estimate and the new information at each update. In this case the forgetting rate (4.12) corresponds to the average case per update, but since it is the geometric mean that is of interest and the geometric mean is heavily influenced by its lowest values - being the cases when the learning rate is higher than expected. This means that the algorithm will tend to learn faster than expected from the analysis of linear regression and may indeed learn fast enough despite a low value of η_0 in (8.9).

It would have been desirable to find a good rule for setting the learning rates such that stability and convergence rate could be somewhat accurately predicted as in the case for linear regression. But unfortunately the learning rate analysis is a lot more complex in the nonlinear case. For matrix factorization with the SDLM method, what can be said is that the global learning rate η_0 should be lower than for a linear parameter and the stabilization parameter γ_2 should be large enough to prevent the denominator to blow up. Aside of that, the best method seems to be to perform trial and error to find a learning rate that is empirically stable, preferably with some margin, yet is high enough to make sure the cost function decreases at an acceptable rate. When evaluating empirical stability, it is important to let the training run for many epochs while monitoring the cost function on the training set as sometimes the errors on the unstable subset of the training data may grow so slowly that the effects aren't noticeable until the error has converged for the stable subset of the training data. This is one of the problems with instability in highly dimensional systems; if training is stopped early there may not be any sign that a small subset of the parameters is misbehaving and deteriorates the results.

Due to the tight stability constraints pushing down the acceptable learning rates for matrix factorization, a matrix factorization model may require a larger amount of epochs than a bias model, which is problematic as the model in itself is more expensive to train. This may even be a reason to avoid the SDLM method for unbalanced nonlinear problems as the spread of learning rates may be very large. The large spread of learning rates used by the SDLM method is what makes sure that parameters converge at the same rate, but if stability puts an upper bound on the learning rates this also means that all parameters are limited by the convergence rate of the parameters for which the least data is available. If there is a large difference between how often parameters are updated and training is very expensive, it may be worth limiting the spread by either using a large value of γ_2 in the SDLM method or by using a more simple method such as the SGD as it allows a higher learning rate to be set for the densely rated parameters without risking the stability of the sparsely rated parameters.

8.6 Hyperparameter Optimization

In contrast to the linear case where regularization made little difference in the results the effect of regularization is much more pronounced when the model complexity is higher. If the feature space is large there may be enough parameters to capture not only the underlying structure, but also most of the noise specific to the data set which degrades the results greatly. Without regularization the results for the movielens dataset and a 50 dimensional matrix model trained for 20 epochs on the residuals from a bias model with an SDLM method using $\eta_0 = 0.02$ for the matrix factorization is an RMSE of 0.9914, which is actually even worse than the plain bias model. If more epochs are added the overfitting of the unregularized matrix factorization gets even more pronounced as there is less implicit regularization arising from the remaining bias of the initial condition. Luckily the code could be run fast enough on the Movielens data set to run the Nelder-Mead simplex optimization to find a good set of regularization costs. With regularization it was found that a higher learning rate could be used, such as $\eta_0 = 0.3$ without causing instability. Using that higher learning rate and regularization costs selected by the Nelder-Mead simplex method the RMSE could be reduced to 0.9346. An important thing to note is that the Nelder-Mead simplex method will have a hard time converging unless the seed is fixed at each iteration such that the feature vectors are initialised the same way each time. If the seeds are different the function evaluations will give inconsistent values during the search effectively preventing the algorithm from terminating as even two evaluations at the same point may yield different values. Based on [15] the feature vectors were initialized to small positive uniformly distributed values using the same seed each time and the Nelder-Mead optimization converged in 121 function evaluations.

The Yahoo! Music data set proved to be much harder to work with. First of all it is 3,000 times larger. The average user has roughly three times more ratings, but the most sparsely rating users have just as few ratings as in the Movielens data set meaning that the learning rates are just as limited. A learning rate defined by $\eta_0 = 0.3$ seemed to be about as high as could be allowed. Using this learning rate and training for three epochs the results had barely had the time to improve compared to the results from the bias model. The alternative was to increase the number of epochs for the matrix factorization despite the increased running time of the algorithm. Increasing the number of epochs for the matrix factorization to 20 increased the running time of the algorithm to 14 hours, which made Nelder-Mead optimization too expensive to perform as its expected time to convergence would be a few months. Instead I decided to first set the regularization costs for the matrix model manually to 10^{-7} as it was on a scale that seemed to be on the somewhat strict side of the reasonable range based on a steady state argument (6.5) had it been a linear parameter. From this position I tested searching nearby grid points and managed to achieve an RMSE of 25.0878 when the regularization costs were 10^{-6}.

8.7 Experimental Results

To get a better overview of how the results improve as more complex models are employed and tuned, see the table below for a summary of the RMSE on the Movielens data. As a benchmark, results from [16] that are using the same models on the same data set are used.

Dataset	Model	RMSE	Benchmark
Movielens	Mean	1.1238	1.1238
Movielens	Bias	0.9615	0.9656
Movielens	Matrix Factorization	0.9346	0.9475

The results on the Movielens data set compare well with the benchmark results indicating that the method is working as it should. There is even some improvement for the matrix factorization results that are responsible for the personalization of the recommendation system. Most of the variance is explained by the basic bias model which shows how important the biases, arising from different item popularities and different users rating patterns, are to capture in order to make good predictions.

The Yahoo! Music data was significantly harder to work with, as a baseline the results by [15] are presented below.

Dataset	Model	RMSE	Benchmark
Yahoo!	Mean	38.2940	38.0617
Yahoo!	Bias	28.1886	26.8561
Yahoo!	Personalization	25.0878	22.9533

Here the results are far from the benchmark. Now there are differences in that their personalization model also used meta data such as time stamps and genre information. Also their mean estimator providing better results than the true mean also suggests that the learning rates may have been tuned to benefit from the arbitrary numbering of the users, a form of hyperparameter overfitting. However, that is not enough to explain the different results. Other groups that have worked with the same data and specified that they did shuffle the data such as [22] have attained results that are still far better than the ones that I managed to achieve. For a plain matrix factorization model [22] attained an RMSE of 23.92 on the test set using an ordinary SGD method trained for 30 epochs.

There are a few differences in how the models were trained. We used the SDLM method, which means that the parameters with a low number of ratings in the data sets can attain a lower bias, but also a higher variance. To compensate for that we scaled our regularization updates (6.3) to regularize these parameters harder. In comparison a plain SGD without scaling the regularization updates may achieve regularization of these parameters due to a limited amount of training, giving an effect similar to early stopping. This difference in how the models are trained and regularized may well be part of the explanation why the models differed in performance.

It could also be that, since our implementation was taking too much time to run, our experiments were limited to a small number of epochs and very little hyperparameter optimization. In fact, the learning algorithm used to run much slower and without extensive code optimization it would have been impossible to perform any experiments at all on the Yahoo! Music data set.

Chapter 9

Implementation Aspects

9.1 Implementation Aspects

Many of the difficulties in working with recommender systems arise from the scale of the data sets used. While the Movielens 100k data set is somewhat straightforward to work with on a modern computer, the Yahoo! Music data set proved to be a challenge computationally. At one point a computation using a plain bias model for the Yahoo! Music data set performed 30 epochs in 183 hours on an Intel 2.67 GHz Core i5 machine with 8 GB RAM. Although that slow running simulation was partially caused by forgetting to turn off the interactive debugger, it was still obvious that improvements needed to be made, especially if the Nelder-Mead method was to even be considered.

An important step in making the training more efficient is to analyse the mechanics of the learning algorithm, see chapter 4, as it provides the means to set the right learning rate from the start. This greatly reduces the number of experiments that have to be performed and also means that the need for many epochs of training to ensure convergence for linear models is reduced. With a better choice of learning rate, the number of epochs needed can be reduced by a factor 10 and still yield better results than what could be attained after weeks of tuning the learning rate experimentally.

However, as tuning the regularization parameters with the Nelder-Mead simplex method requires hundreds of function evaluations and the matrix updates makes training much more time consuming, it is crucial to also speed up the code as well to make sure all needed experiments can be performed on time. To understand how changes in the code can result in faster execution speed we need to discuss the Python programming language a little bit more in depth.

9.2 Python Code

Python is a high level language, that uses dynamic typing and was designed with readability in mind. This means Python code is fast to write and generally quite easy to understand. As an example, consider the code example in listing 1 for predicting how a given user would rate a given item.

```
def estimate_rating(user, item, data):
    estimate = data.bias + user.bias + item.bias
    if MATRIX_MODEL:
        estimate += numpy.dot(user.features.T,item.features)
    return estimate
```

Listing 1: A basic Python implementation of code to generate a prediction of how a user would rate an item.

The code is quite readable, the estimated rating is calculated by summing up the bias terms arising from the data, the specific user and the specific item. If a matrix factorization model is used the inner product between the feature vectors is calculated using a function called *dot* that is imported from the numeric Python package NumPy. The inner product is added to estimate and the result is returned to the calling function.

The code is however not optimized for speed or memory and even the innocent looking second row translates to many rows of low level code. First, the program must verify that the Python objects *data* and *user* have attributes called *bias* and increase those attributes' reference counts, then it adds those variables together, creates a reference to the sum and clears the reference count of the bias terms before repeating the process for the item bias term and creating a reference to the new result. For most of those intermediary operations there is also an associated exception handler that makes sure exceptions are thrown if any of the checks fail.

Many of these operations are not really necessary for the code to work as intended but the program will perform these checks anyway in case the programmer has made a mistake. Not having to write all this low level code is one of the reasons Python code is highly readable and easy to write. Not having the control to turn off the checks that are unnecessary and optimize the memory allocations is a reason that a low level program written in for example C may perform much faster. That being said, there are ways to speed up Python programs while maintaining many of Python's strong points.

9.3 Profiling Python Code

When trying to speed up code there is one principle that should always be followed and that is to never optimize without profiling. Profiling is done with a profiler, which is a tool that you run together with the code to learn where in the code most of the execution time is spent. Without a profiler you can not tell which parts of the program are worth trying to optimize and whether the changes you make to the code have the desired effect. Most Python installs include a profiling program called cProfile. The easiest way to profile a program named recommender.py is to set up a script such as the one in listing 2, which imports the profiler, cProfile, and a module called pstats that gives better output from the profiler. The program is then run with the profiler attached and the 10 most important posts are printed to terminal.

```python
import cProfile
import pstats
import recommender

cProfile.runctx("recommender.run()", globals(), locals(), "Profile.prof")
s = pstats.Stats("Profile.prof")
s.strip_dirs().sort_stats("time").print_stats(10)
```

Listing 2: A simple script for running the Python program recommender with a profiling tool and printing the top ten operations that the program spent time on.

9.3.1 Profiling Results

By profiling my code we can tell that most of the time is spent doing simple arithmetic operations. For a simple experiment including setup, training a basic bias model one epoch and then validating the results it is found that for the early implementation in listing 1 45 % of the time is spent updating bias terms, 25 % is spent estimating ratings and 20 % is spent calculating the RMSE at the end. Of the remaining 10 % about half is spent in the main loop performing input/output and the remainder is spent at setup.

Establishing which operations are expensive in turn means that we can prioritize which parts of the code to scan for possible improvements. Rather than spending a day trying to optimize how data is read from the file we can focus on the parameter updates and spot minor inefficiencies such as unnecessary explicit type casts which on their own costed more than every file access costed in total.

9.4 Code Optimization

Having established which operations were costly means that some low hanging fruit can be collected. Loops utilising lists of indices can be replaced by loops using iterators, explicit type casts can be removed and some of the global variables can be replaced by locals. It is also good practice to refactor the code to get rid of lines of code that had outlived their usefulness. However, one reason the code is slow is that it uses up almost all available memory on the computer it was being run on. To reduce the memory footprint we can change the data structures used for storing parameters. Rather than lists of user objects with encapsulated parameter values, these parameter values themselves being Python objects with their own memory overhead, we can use NumPy arrays for parameter storage. NumPy offers several highly optimized linear algebra routines but the main reason we would like to use it is that it offers improved memory efficiency as the arrays for many purposes are treated as one object containing many values instead of many small objects with individual memory overhead. While working on memory efficiency we can also change the file format of data files to HDF5 files while only maintaining the data fields that are actually used. This can be achieved with the *pytables* module and reduces the disc space required by the data files from 8 GB down to 2.4 GB.

Making the application more memory efficient means that we no longer have to struggle against insufficient disk space and memory errors. The code would also runs more

```
import numpy
BIAS = 3  # index of bias terms in user and item.
MATRIX_MODEL = 1
estimate_rating(mean, user, item):
      est = mean + user[BIAS] + item[BIAS]
      if MATRIX_MODEL:
            est += numpy.dot(user[FEATURE_1:FEATURE_END],
                                        item[FEATURE_1:FEATURE_END].T)
      return est
```

Listing 3: Another Python implementation of code for estimating a rating. Storing all parameters in a NumPy array makes the code more memory efficient which can be very important if the computer does not have much RAM.

smoothly, especially long running simulations on the Yahoo! Music data set becomes much faster with these modifications. Now that it is possible to run the code with matrix factorization without getting memory errors we can profile again to measure which operations are the largest time sinks when the full model is used.

Method	time/call (Python)	time/epoch
estimate rating	4 μs	17 m
bias update	5.3 μs	24 m
matrix update	60 μs	4 h 10 m

These three methods amounted to over 90 % of the execution time, and it would be of great value if these methods could be made even faster. The next step in the optimization procedure is then to see if these methods can be made faster by using a tool to generate and run optimized C code.

9.5 Cython

Cython is a programming language that is intended to make it easy for a Python programmer to write extension modules in C. With Cython you can take a piece of Python code and compile it to C code. It also allows the programmer to add static type declarations and decorators telling the compiler that it does not, for instance, need to perform type checking inside functions that the programmer has provided declarations for. This means that you can keep the code structure from your Python program, the only difference is that you have to compile the code before running it.

If there is a function that you would like to optimize, such as the code for estimating the rating of a user, you can take that code and compile it with the *cython -a* switch. This creates an HTML file where lines are color coded depending on how well they translated into C code. Lines with a large overhead are coloured bright yellow, and lines with low or no overhead are light yellow or without overlay. Each line of Cython code in the HTML file can be clicked to examine the generated C code. If there for instance are many type checks in the C code for variables with known type you can speed up the code by providing type declarations. A basic Cython version of the code for estimating a rating is seen in listing 4.

```
import numpy
cimport numpy
ctypedef numpy.float64_t DTYPE_t
DEF BIAS = 3  # index of bias terms in user and item.
DEF MATRIX_MODEL = 1
cdef float estimate_rating(
            float mean,
            numpy.ndarray[DTYPE_t, ndim=1] user,
            numpy.ndarray[DTYPE_t, ndim=1] item
            ):
      cdef float est = mean + user[BIAS] + item[BIAS]
      IF MATRIX_MODEL:
            est += numpy.dot(user[FEATURE_1:FEATURE_END],
                                        item[FEATURE_1:FEATURE_END].T)
      return est
```

Listing 4: A Cythonized version of the Python code for estimating a rating. Cython allows you to use compiler directives and static type declarations to make the code run faster.

The code example in listing 4 highlights some of the features of Cython. The first difference lies in the *cimport* of NumPy which allows C typing for NumPy arrays. Next is a typedef which allows types to be predefined in case you need to change the variable type at a later stage. There are also precompiler instructions in the form of *IF* and *DEF*, which are evaluated by the precompiler and inserts the correct code before the code is compiled, meaning that the *IF* is never evaluated in the loop. This also means that you do no longer have to choose between a global variable and a magic number, which makes it easier to write good code.

The *cdef* at the function declaration means that the function calls from within C will have reduced overhead. Most important are the type declarations of the local variables *mean*, *user*, *item* and *est* that tells the final C code that it will not have to bother with type checks for the typed variables.

Looking at the annotated HTML still shows most of the lines in yellow, but running the profiler shows that the method is much faster already. The same procedure is done for the bias updates with similar results. The generated C code tells us that there is still some overhead resulting from checking that the accessed array indices are legal and some overhead as the function is initialised and returns, but overall the improvements are substantial.

The matrix updates that are responsible for roughly 75 % of the running time of the code, counting setup and evaluation, can also be made substantially faster by providing type declarations. Still, there is a lot more to do to improve the speed of this method. One thing to note is that the C code ends up performing many expensive slicing operations and function calls to Python functions in the NumPy library. While the NumPy functions are highly optimized for large linear algebra problems, the inner products in question are comparably small and a simple for loop may be expected to translate better into C code as calls to Python functions from C are expensive. However, this leads to lots of array accesses which comes at the price of many checks of the array indices to make sure they are positive and within the range of the arrays.

In order to speed up array access we can use decorators. Decorators are instructions to the compiler that for example tells the compiler that all indices of an array that are accessed are legal to access which means no code has to be provided to check for invalid indices. This can speed up array operations significantly, but if the programmer makes a mistake it can results in serious errors as the code may attempt to access memory that should not be available to it. The use of decorators is demonstrated in listing 5 which runs in only 0.92 μs per call. Checking the C code it is not very surprising that this code runs faster. The C code generated for the inner product was reduced from 30 rows of C code, including several expensive calls to Python functions, to five rows of pure C code.

```
import numpy
cimport numpy
ctypedef numpy.float64_t DTYPE_t
DEF BIAS = 3  # index of bias terms in user and item.
DEF MATRIX_MODEL = 1

@cython.boundscheck(False)
@cython.wraparound(False)
cdef float estimate_rating(
      float mean,
      numpy.ndarray[DTYPE_t, ndim=1] user,
      numpy.ndarray[DTYPE_t, ndim=1] item
      ):
cdef float est = mean + user[BIAS] + item[BIAS]
IF MATRIX_MODEL:
      for k in range(PVEC_START, PVEC_STOP):
            est += user[k]*item[k]
return est
```

Listing 5: An optimized version of the Cython code for estimating a rating. By replacing the function call to numpy.dot with a for loop and adding decorators the code could be made significantly faster.

With this information the matrix factorization can be improved by using a for loop and adding decorators which brought its execution time down to $4.0\mu s$ per call. A summary of the performance gains for the methods that is attained for some of the most important methods can be seen below.

Method	Time/call (Python)	Time/call (Cython)	Speed up
estimate rating	4 μs	0.9 μs	4.4 $\times$
bias update	5.3 μs	0.8 μs	7 $\times$
matrix update	60 μs	4 μs	15 $\times$

With this level of code optimization the training of a 50 dimensional matrix factorization model for 20 epochs on the Movielens data finishes in less than 20 seconds. Training such a model on the much larger Yahoo! Music data set for three epochs finishes in roughly three hours, which is still somewhat slow as it would have been desirable to use more epochs. But if the Nelder-Mead method is to be used it requires dozens of function evaluations to converge which means it's necessary to keep execution times down such that the hyperparameter optimization can be expected to finish in time. At this stage of optimization the tight loops are running almost pure C code except some Python overhead when entering or leaving a function. The function that is currently using most of the execution time is the update manager that reads data from file and processes it for the various update methods. Optimizing this method is harder as it relies on several Python packages that would require much work to replace.

9.6 Scaling it up?

The Yahoo! Music data set is a large data set, but there are much larger data sets out there. The Million Song Dataset [23] for example is slightly larger in amount of songs, but it also comes packed with tons of metadata that makes it much bigger. That data set comes in a 300 GB HDF5 file which is roughly 40 times the disk size of the Yahoo! Music data set before compression. Commercial streaming services face even bigger challenges. Not only are the amounts of users and items typically much bigger, Spotify for example has over 24 million users and about as many tracks. This means that a commercial scale algorithm has to be able to process data sets more than a hundred times larger than what we've been doing in this thesis. Commercial recommendation systems may also need to be able to produce recommendations in real time and although many computations can be done in advance it still puts additional demands on the system.

To scale up the algorithms used in this thesis for a problem of that size would require a lot more computing power and memory. Realistically a distributed approach would probably need to be used which means that the algorithms used would have to be replaced by versions that are better suited for a parallel implementation. In their standard forms neither stochastic gradient descent nor Nelder-Mead search are suited for parallelism but there are modified versions that use some measure of parallelism. See for example [24] for a parallel implementation of the Nelder-Mead method, [25] for a parallel version of the stochastic gradient descent method and [12] for a discussion of higher order mini-batch methods that scale well and can be distributed to multiple cores or GPUs.

Chapter 10

Summary and Conclusions

10.1 Summary

In this thesis the problem of predicting people's preferences for different media items based on historical ratings of such items has been studied. The methods studied are the stochastic gradient descent method and the stochastic diagonal Levenberg-Marquardt method both minimizing a cost function (3.1) on the whole ensemble of training data by only processing one training example at a time which makes them very scalable. These stochastic descent methods were known to require much manual tuning to get working at their full potential which is problematic as experiments can be very computationally demanding for large data sets.

It was found that the learning rates could be associated with the attenuation of initial conditions and the stability of the parameter estimates. This information can be used to set the learning rate with much less need for trial and error, which can save many hours of work. It is found that the stochastic diagonal Levenberg-Marquardt method distributes the learning rates of the parameters such that the attenuation of the initial conditions is the same for all parameters. For linear regression the stability does not depend on the parameter values, which means that the learning rates and number of epochs can be tuned to match given specifications on attenuation and stability. Experimentally it was found that the algorithm behaved as expected for two different data sets of vastly different scales, which indicates that the results should most probably generalize well to similar problems in linear regression.

For the nonlinear regression that is needed to capture the interaction between users and items it was found to be much harder to analyse stability and attenuation. This is because the evolution of parameter values depends on current parameter values. What can be concluded is that the optimization process will in general be unstable for some parameter values. This in combination with the nonlinear stochastic evolution of parameter values makes it very hard to predict whether the algorithm will remain stable or not. Experimentally it was found that the learning rates had to be set to lower values than for the corresponding linear models to remain stable when using the SDLM method, which in turn lead to the demand for long running simulations to see any improvements over the pure bias models.

While the basic bias models had fairly low complexity relative to the large amount of training data available, the matrix factorization models have a very high complexity. For the Movielens data the matrix factorization model with 50 dimensions have more model parameters than there are training examples in the data set. To prevent overfitting the cost function was extended by adding penalty terms proportional to the squared magnitude of the model parameters. To ensure that the optimization procedure stayed true to the selected cost function the regularization updates were scaled by the parameter frequencies to avoid imposing a higher cost on frequently occurring parameters. While many practitioners omit the rescaling of the regularization costs since it makes it harder to control the stability of the learning algorithm, we stayed true to the cost function to see what that would buy us in return. The values of the regularization parameters were selected using the Nelder-Mead simplex method, which performs many function evaluations to find good values of the regularizaion costs.

To afford the many function evaluations needed to optimize the regularization costs it proved necessary to make the initial python code run faster. By profiling the code, critical code segments could be identified and optimized using Cython. The speed gains were attained by providing static type declarations, removing global variables, replacing calls to python libraries with simple loops and adding decorators such that Cython could generate optimized C code that could perform the tasks much faster than the original Python code. While the original Python code had left much room for optimization the final transition to Cython made the resulting code roughly one order of magnitude faster, which was crucial in order to perform all the necessary experiments to debug the code and generate results.

When it came to predicting a user's ratings of movies in the Movielens 100k data set the resulting predictions on the test set had an RMSE of 0.9346 which compares well with available benchmark results [16]. For the much larger and sparser data set of music ratings it proved much harder to get results that compared well with the available benchmarks, the best that could be attained was 26.6859 which is quite far from the results of [15] that managed to bring the RMSE down to 22.9533 using a matrix factorization model. While the models for the Yahoo! music data set were not identical in that they used a more extensive modelling of bias terms and made use of more metadata, even their intermediary results using simpler models, such as the basic bias model or the constant predictor, outperformed the corresponding intermediary results that we could attain with our experiments. The intriguing results by [15] for the constant predictor model that outperformed the true mean could be an indicator that their results suffers from some amount of hyperparameter overfitting. More importantly, they had better means of evaluating and comparing many different hyperparameter settings with many epochs of training, which we could not, as our code despite the code optimization was still too slow.

10.2 Further Studies

In any further studies involving large data sets we would recommend making it a priority to make the training procedure run fast enough to perform experiments of desired precision at a low cost. As such it would be of interest to see how to best implement algorithms for training recommender systems in a distributed environment. If the algorithms can be made faster by adding more nodes to a cluster that would be a great benefit as it frees us of the time constraints imposed by how fast the training can be performed on a single computer. If training was fast it could for example be used to determine experimentally whether the the SGD method or the SDLM method gives best results when the amount of epochs is allowed to be large. It would also be of interest to see if the batch methods can handle a problem of this scale and how such a method would compare in speed, ease of tuning and generalization power of the results.

It would also be of great value to find a method for learning of scaled nonlinear regression problems that was less reliant on hyperparameter optimization. With such a method one would not have to perform as many experiments which could save lots of time and effort.

Bilaga A

Populärvetenskaplig sammanfattning

Aldrig förut har film och musik varit så lättillgänglig som idag. Strömningstjänster som Spotify och musiknedladdningstjänster som iTunes och Amazon MP3 ger sina tillgång till över 20 miljoner låtar vardera. Med så mycket tillgänglig musik är vi helt beroende av rekommendationer för att hitta ny musik. Det finns helt enkelt för mycket musik för att kunna lyssna igenom allt. Musikrekommendationer kan finnas i många skepnader - från topplistan som spelas på radio till tips från en kompis som hört ett bra band på en festival. De stora musiktjänsterna använder sig ofta av ett automatiserat system som ger rekommendationer baserat på den information de har om hur du och andra användare brukar använda tjänsten. Är de automatiska rekommendationerna bra så kan det leda till fler köp och nöjdare kunder varför företagen lägger stor vikt vid att utveckla så bra rekommendationssystem som möjligt.

I detta examensarbete har en metod för att förutsäga vad en given användare tycker om en given låt, artist eller genre baserat på vad den satt för betyg på den musik den lyssnat på hittills studerats. Metoden är en typ av övervakad inlärning vilket innebär att en dator tränas på en stor mängd användardata där vi redan vet vilket betyg användaren satt. Om modellen förutsäger för höga betyg kan den då justeras baserat på denna träningsdata så att dess förutsägelser stämmer bättre överens med träningsdata. Eftersom en typisk musiktjänst har miljontals användare finns det väldigt mycket data att tillgå. Därför hade det varit väldigt tidskrävande att gå igenom all data innan någon uppdatering av modellen sker. Istället används ofta så kallade stokastiska optimeringsmetoder där man tar ett slumpmässigt utvalt träningsexempel i taget och uppdaterar modellen baserat på det exemplet istället. På så vis kan algoritmen hinna med miljontals uppdateringar på den tid som den annars bara hade hunnit med någon enstaka. Var och en för sig är uppdateringarna som är baserade på ett enstaka exempel väldigt osäkra i jämförelse med om alla exempel behandlats på en gång, men om tillräckligt många uppdateringar görs tenderar osäkerheten att jämna ut sig med tiden.

När man använder den här typen av inlärningsalgoritmer måste man göra en avvägning hur mycket man ska uppdatera modellen för varje träningsexempel i förhållande till hur mycket man vill behålla av den modell man redan tagit fram. Ett aggressivt uppdateringsschema innebär att modellen anpassar sig snabbt, men att den slutgiltiga modellen blir mer osäker. Om man istället gör väldigt försiktiga uppdateringar av modellen blir modellsäkerheten större, men det behövs mer tid att träna upp den. Det som särskilt studerats är en specifik metod för hur man väljer hur stora uppdateringar som ska göras vid varje träningsexempel. Metoden innebär att de delar av modellen som det är svårt att få information om, till exempel en låt som väldigt få användare lyssnar på, uppdateras aggressivt när chansen väl ges. Modellen för populära låtar kan uppdateras mer försiktigt då det i regel finns mycket träningsdata för detta.

Det som vi kunde komma fram till är att den här typen av inlärningsschema är lätt att anpassa för enklare modeller som beskriver hur olika användare generellt sett betygsätter låtar och vilka betyg låtarna i allmänhet brukar få. Däremot visade det sig vara avsevärt svårare att hitta en balans mellan snabbhet och tillförlitlighet när mer komplexa modeller som försöker fånga in information om användarnas mer specifika musiksmak och vilka låtar som liknar varandra skulle tränas upp. Detta beror på att den typ av modeller som krävs för att fånga upp denna typ av mönster beter sig väldigt olika under olika stadier av inlärningsprocessen, vilket i sin tur gör det svårt att anpassa ett inlärningsschema som fungerar bra från början till slut. Det kan till exempel hända att den övre gräns för hur aggressiv inlärning som går att använda ändras under inlärningens gång. För att kompensera för detta känslighet behövdes en försiktigare inlärningstakt användas rakt igenom, vilket i sin tur ledde till att träningen blev väldigt tidskrävande. Detta är ett problem om systemet ska kunna användas i praktiken eftersom ett rekommendationssystem inte bara måste vara träffsäkert för att vara till nytta, det måste också kunna köras snabbt i bakgrunden så att de som använder systemet inte märker av alla tunga beräkningar som ligger bakom varje rekommendation.

Bibliography

[1] Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems. *IEEE Computer*, 42(8):30–37, 2009.

[2] J. Bennett and S. Lanning. The Netflix Prize. In *Proceedings of KDD Cup and Workshop 2007*, 2007.

[3] W.T. Glaser, T. B. Westergren, J.P Stearns, and J.M Kraft. Consumer Item Matching Method and System, 2006.

[4] Movielens 100k. URL `http://grouplens.org/datasets/movielens/`. Accessed 2014-06-12.

[5] Yahoo! Music user ratings of musical tracks, albums, artists and genres, v 1.0. URL `http://webscope.sandbox.yahoo.com/`. Accessed 2014-06-12.

[6] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The yahoo! music dataset and kdd-cup'11. In *Proceedings of KDDCup 2011*, 2011.

[7] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating Collaborative Filtering Recommender Systems. *ACM Transactions on Information Systems (TOIS)*, 22(1):5–53, 2004.

[8] L.-C. Böiers. *Lectures On Optimization.* KFS AB, 2004.

[9] L. Bottou. Online Learning and Stochastic Approximations. In D. Saad, editor, *Online Learning and Neural Networks.* Cambridge University Press, 1998.

[10] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. *Neural Networks: Tricks of the Trade*, chapter 1 Efficient Backprop, pages 9–50. Springer, 1998.

[11] L. Bottou and Y. LeCun. On-line Learning for Very Large Datasets. *Applied Stochastic Models in Business and Industry*, 21:137–151, 2005.

[12] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng. On Optimization Methods for Deep Learning. In *Proceedings of the 28th International Conference on Machine Learning*, 2011.

[13] L. Bottou and N. Murata. Stochastic Approximations and Efficient Learning. In *The Handbook of Brain Theory and Neural Networks.* MIT Press, 2002.

[14] J. Nocedal. Updating Quasi-Newton Matrices with Limited Storage. *Mathematics of Computation*, 1980.

[15] G. Dror, N. Koenigstein, and Y. Koren. Web-scale Media Recommendation Systems. *Proceedings of the IEEE*, 100:2722–2736, 2012.

[16] MovieLens 100k Benchmark Results, February 2013. URL `http://www.recsyswiki.com/wiki/MovieLens_100k_benchmark_results`. Accessed 2014-06-12.

[17] A. E. Hoerl and R. W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12:55–67, 1970.

[18] T. A. Johansen. On Tikhonov Regularization, Bias and Variance in Nonlinear System Identification. *Automatica*, 33:441–446, 1997.

[19] A. M. Urmanov, A. V. Gribok, J. W. Hines, and R. E. Uhrig. An Information Approach to Regularization Parameter Selection Under Model Misspecification. *Inverse Problems*, 18:1–22, 2002.

[20] W. Finnoff, F. Hergert, and H. G. Zimmermann. Improving Model Selection by Nonconvergent Methods. *Neural Networks*, 6:771–783, 1993.

[21] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7:308–313, 1965.

[22] M. Jahrer and A. Töscher. Collaborative Filtering Ensemble. *Journal of Machine Learning*, 2012.

[23] T. Bertin-Mahieux, D. P.W. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[24] D. Lee and M. Wiswall. A Parallel Implementation of the Simplex Function Minimization Routine. *Computation Economics*, 30(2):171–187, September 2007.

[25] R. Gemulla, P. J. Haas, E. Nijkamp, and Y. Sismanis. Large Scale Matrix Factorization with Distributed Stochastic Gradient Descent. *KDD*, 2011.