Eötvös Loránd University

Faculty of Natural Sciences

Online Recommendation Systems

MSc Thesis

Domokos Miklós Kelen

Applied Mathematics MSc

Advisor:

András Benczúr Ph.D.

Department of Operations Research

Budapest, 2016

Köszönetnyilvánítás

Szeretném megköszönni témavezetőimnek, Benczúr Andrásnak és Pálovics Róbertnek, hogy iránymutatással és fáradhatatlan segítőkészségükkel támogatták ezen munka elkészülését. Köszönettel tartozom továbbá a SZTAKI Informatikai Kutatólaboratóriumának, dolgozóinak, és mindenki másnak is, aki támogatásával vagy segítségével hozzájárult a munkámhoz.

Contents

Introduction

The goal of a recommender system is to predict preference relations between different entities, usually users and some kind of product, such as music, movies, websites, but even scientific papers or news articles. They provide personalized recommendations by analyzing patterns in the users' interest. Their importance in customer facing applications is very high and ever growing, since it is their responsibility to guide the customer to products that he or she is most likely to consume. Recommender systems are used by millions of people every day.

The subject of recommender systems research has gained a lot of popularity with the Netflix prize challenge, which was a competition held between the years 2006 and 2009. The movie rental company Netflix announced a grand prize of $1,000,000 to be won by the first research team completing the challenge: to outperform Netflix's own algorithm by 10%. For this purpose, they provided a dataset consisting of about 100million records. The prize was won in 2009 by team *BellKor's Pragmatic Chaos* (a merger of multiple competing teams). The winning algorithm united multiple strategies and algorithms in an ensemble method, which was able to beat the baseline ratings by 10.06%.

In this thesis, we describe fundamental concepts, problem settings, algorithms and evaluation methods related to recommender systems. Our research was aimed at improving ranking list predictions, building upon the results of Harald Steck, presented in his 2015 paper *Gaussian Ranking by Matrix Factorization.* More specifically, we attempted to apply the ideas described in the paper for online recommendation. In this context, *online* refers to the way the algorithm processes the data.

In Section 1, we summarize necessary concepts related to recommender systems. In Section 2, we describe the popular matrix recommender system model, the matrix factorization model, and different algorithms for training this model. In Section 3, we summarize the results of Steck et al., and present our own approach to applying the ideas for online recommendation. Finally, in Section 4, we describe our experiments and their results.

Chapter 1

Recommender systems

Recommender systems have been the subject of extensive research in the past decade. Over the years, many different approaches have emerged. Various real life examples inspired different problem settings and mathematical models, and many methods have been developed to solve these recommendation tasks. In this chapter, we summarize some of these models and the necessary terminology.

1.1 Ranking and rating prediction

Recommender systems deal with the relationships between the elements of two distinct sets: *users* (N) and *items* (M). In a typical real life scenario, the users are customers and the items are products. The most common goal is then to predict preference relations between users and items. There are multiple ways to model user-item relations. Here we describe two of the most important tasks.

The *rating prediction* model assumes that there exists a preference relation for each $(u, i) \in N \times M$ user-item pair, and also that these relations are quantified, e.g. on a linear 1-5 scale. Lower rating typically means less preference. However, only part of these data is known, and the job of the recommender system is to predict the unknown preference relations between users and items. We denote the observed ratings between users and items with r_{ui}, and the predicted rating of user u of item i with $\bar{r}_{ui}$. This model is well suited for applications where we need to provide the user with a personalized predicted score for *any* item. An example is the Netflix web site, where such predictions can be helpful for a user for deciding whether to watch a movie or not. However, this is not the only approach possible.

In many situations, it is required of the algorithm to provide the user with a toplist, consisting of the most relevant elements of the item set. We call this the *ranking prediction task*. Here it is not necessary to assume that the user-item preference relations can be quantified, unlike with rating predictions. Instead we assume, that the items with ratings form a totally ordered set: for each user u there exists a relation $<_u$ over the item set, such that $<_u$ is *transitive*, *antisymmetric* and *total*. Our goal is then to predict the K greatest elements of the item set for each user. In many cases, ranking predictions are computed by employing rating prediction first, and taking the K items with the highest predicted rating. It has been shown however, that utilizing the properties of the above model can result in accuracy improvements over the rankings predicted by classical rating prediction models [1].

Note that when a rating prediction model is used, the assumption of antisymmetry doesn't necessarily hold. Additionally, most of the time observed explicit data about the real rankings of items is not available, so the evaluation metrics must be chosen accordingly.

A closely related notion is *top K recommendation*: in this case, instead of predicting the rank of all the items, the prediction model is only required to predict the top K elements of the ranking list.

We discuss these topics further in Sections 1.5, 2.1 and 4.1.

1.2 Collaborative filtering

Two different types of predictive models can be identified based on the types of the datasets used when examining recommender systems: *content filtering* and *collaborative filtering* based approaches.

Content filtering methods try to build a profile for each entity (user or item) in the system based on contextual information. For example, if the items are movies, such contextual information may include the genre of the movie, actors, critical ratings. User profiles may include things such as age or demographic information [2].

On the other hand, collaborative filtering is the technique of recognizing patterns in recorded interactions between users and items, and making predictions based on this information. Collaborative filtering methods differentiate users / items only by their relation to each other, using no additional context information. They try to make predictions based on the assumption that if users behaving similarly in the observed data, they will behave similarly in the data to be predicted.

$$\begin{array}{c|ccccccc}
 & i_1 & i_2 & i_3 & \dots & i_{m-2} & i_{m-1} & i_m \\
\hline
u_1 & & & & & & 1 & \\
u_2 & & 1 & & & 5 & & 3 \\
u_3 & 3 & 1 & & & ? & & 4 \\
\vdots & \vdots & & & \dots & & & \\
u_{n-2} & & & & & & & \\
u_{n-1} & & & 2 & & & & \\
u_n & & & & & & 3 &
\end{array}$$

Figure 1.1: An illustration of observed and missing rating data between users $u_1, \dots, u_n$ and items $i_1, \dots, i_m$. One could try to „guess” the value of the red question mark based on similar rows of the matrix - most likely the one belonging to u_2 in this case.

There are multiple popular algorithms and models for collaborative filtering, including neighborhood based models and matrix factorization models.

Neighborhood based recommendation

The idea of the neighborhood based approach, also known as *k nearest neighbor* (kNN) is that similar users like similar items. It works by defining some kind of similarity measure between users or items and utilizing the observed data about similar users and items. They are generally liked because they are intuitive and they are simple to implement. Also they are preferred because a clear explanation can be provided to the user for the predictions, which can be useful in a real world application.

There are two main approaches when dealing with neighborhood based models, user based and item based methods. User based neighborhood methods try to make predictions based on how similar users behave, for example by taking the k most similar users and averaging their ratings on the selected item, possibly weighted by the similarity between the users, i.e.

$$\overline{r}_{ui} = \frac{\sum_{v \in N_k(u)} S_{uv} \cdot r_{vi}}{\sum_{v \in N_k(u)} S_{uv}},$$

where $N_k(u)$ denotes the k most similar users to user u and S_{uv} denotes the similarity between users u and v.

Another natural approach is to base the prediction of r_{ui} on how user u relates to items similar to i, i.e.

$$\overline{r}_{ui} = \frac{\sum_{j \in N_k(i)} S_{ij} \cdot r_{uj}}{\sum_{j \in N_k(i)} S_{ij}}.$$

Here, $N_k(i)$ denotes the k most similar items to item i and S_{ij} denotes the similarity between users i and j. Since in a real system the number of users is significantly higher than the number of items, the user-oriented approach is usually considered slower and less accurate [3].

Multiple similarity measures are commonly used for this purpose, usually operating over the corresponding rows (or columns) of the user-item rating matrix to calculate the similarity between users (or items). It is also customary to normalize this data before using it, for example by correcting for user-average ratings or item-average ratings. The similarity measures used include the Manhattan, Euclidean, Minkowsky or Hamming distance, but the most popular ones are probably the Pearson distance (Pearson correlation coefficient) and the closely related cosine similarity [3]:

$$\text{Manhattan-distance}_{uv} \overset{\text{def}}{=} \sum_{i \in M} |r_{ui} - r_{vi}| \tag{1.1}$$

$$\text{Euclidean-distance}_{uv} \overset{\text{def}}{=} \sqrt{\sum_{i \in M} (r_{ui} - r_{vi})^2} \tag{1.2}$$

$$\text{Minkowsky-distance}_{uv} \overset{\text{def}}{=} \left(\sum_{i \in M} (r_{ui} - r_{vi})^p \right)^{\frac{1}{p}} \tag{1.3}$$

$$\text{Hamming-distance}_{uv} \overset{\text{def}}{=} \sum_{i \in M} I(r_{ui} \neq r_{vi}) \tag{1.4}$$

$$\text{cosine-similarity}_{uv} \overset{\text{def}}{=} \frac{\sum_{i \in M} r_{ui} r_{vi}}{\sqrt{\left(\sum_{i \in M} r_{ui}^2\right) \left(\sum_{i \in M} r_{vi}^2\right)}} \tag{1.5}$$

$$\text{Pearson-distance}_{uv} \overset{\text{def}}{=} \frac{\sum_{i \in M} (r_{ui} - \mu_u)(r_{vi} - \mu_v)}{\sqrt{\left(\sum_{i \in M} (r_{ui} - \mu_u)^2\right) \left(\sum_{i \in M} (r_{vi} - \mu_v)^2\right)}}, \tag{1.6}$$

where I is the indicator function, M is the set of all items and μ_u and μ_v are the user-average ratings: $\mu_u = \frac{1}{|M|} \sum_{i \in M} r_{ui}$ and $\mu_v = \frac{1}{|M|} \sum_{i \in M} r_{vi}$. These similarities can be defined analogously for item-similarities.

It is also possible to use context-based information to better approximate the similarity of users or items, or to use information acquired through other learning methods [4], for example singular value decomposition (discussed in detail in Section 2.2.1).

We discuss matrix factorization, another dominant collaborative filtering method in Section 2.

1.3 Offline and online recommendation

When training recommender systems based on collaborative filtering, the available dataset is often represented as a set of user-item pairs and the corresponding ratings. The usual approach is then to split the available data into two, the *training set* and the *test set*. This can be done based on time information attached to the records, or randomly. The training data is used to train the models, and the predictions are evaluated on the test set to measure the accuracy of the system. Often an additional third set, the *validation set* is necessary, used for evaluating the model while in training. This methodology is called *offline* recommendation.

In a realistic example, instead of a set, the feedback data is usually presented on a linear timeline and the recommender system is required to make predictions between each two pieces of new information, in a time sensitive manner. This is called *online* recommendation. In such cases, it is an essential requirement of the recommender system that it is able to retrain its prediction model incrementally, on a record-by-record basis.

It is possible to employ an offline recommendation model for online recommendation by processing the data in batches, treating the some or all of the records before the current one as the training set, and not-yet-known records as the test set. This approach might not be computationally feasible.

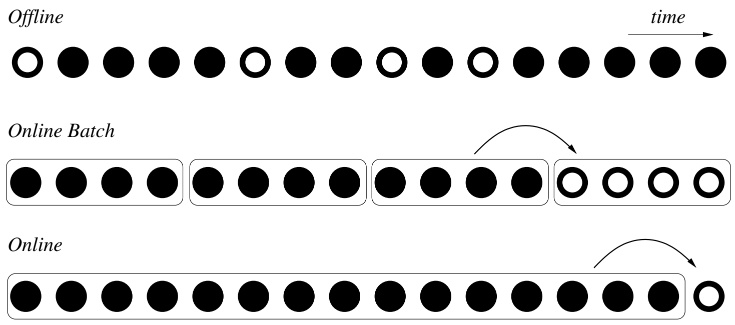


Figure 1.2: Offline recommendation, online recommendation with batch processing of the data, and online recommendation.

Another interesting aspect of online recommendation is that since the popularity of certain items may change over time, both in general and in relation to specific users. Because of this, it can be beneficial to consider the recency of the feedback data when training the model. This essentially means that if a piece of information is more recent, it is more reliable, and should weight more in our prediction model.

1.4 Implicit and explicit recommendation

In the classical setting, e.g the Netflix challenge, the observed r_{ui} records are ratings of items by the users on a linear scale. This kind of data is called *explicit feedback* and can be used well to make rating predictions. However, in some cases, the actual ratings of the items are not available. Consider for example a music streaming service that logs the list of songs a user listens to. This data has no explicit information about ratings, yet we can most certainly deduce some information about the users' preferences. This kind of data is called *implicit feedback*. For simplicity's sake, we will call implicit interactions (implicit) ratings. The number of interactions between users and items are often discarded, meaning implicit ratings are 1 or 0 valued.

Similarly, in some cases, we are trying to predict the likelihood of interactions between users and items, without these interactions necessarily indicating preference relations. We call this the *implicit recommendation task* (as opposed to *the explicit recommendation task*).

Implicit recommendation and explicit recommendation are closely related concepts. It is plausible to build a recommendation system by viewing implicit interactions as ratings on a 0-1 scale, and as such it can be viewed as a special case of explicit recommendation. However, implicit and explicit feedback are fundamentally different in certain ways: while explicit feedback carries information about positive and negative preferences as well, negative feedback is (usually) inherently nonexistent in the case of implicit feedback, thus all user-item relations are either positive or unknown. In the remainder of this thesis, we call the known implicit user-item relations positive ratings, and the not yet known elements negative ratings.

It can be beneficial to study the implicit recommendation task on its own for performance reasons and other considerations, as the fact that R consists of $0-1$ values can be used to construct methods that exploit this property of the matrix. One can also attempt to use implicit data for explicit recommendation, assuming the implicit feedback has some indication of user preferences. In the example of users listening to songs, there is a clear indication of preference.

1.5 Evaluating prediction accuracy

There are multiple widely used metrics for evaluating recommender systems. Here we describe the most commonly used ones.

1.5.1 Evaluating rating prediction

The standard way to measure the accuracy of rating predictions is to take the *root mean squared error* (RMSE) of the predictions over the known values. The RMSE value of a set of predicted ratings H is given by the formula

$$\mathrm{RMSE}_H \overset{\text{def}}{=} \sqrt{\frac{1}{|H|} \sum_{(u,i)\in H} (\overline{r}_{ui} - r_{ui})^2}, \tag{1.7}$$

where r_{ui} is the true rating of user u on item i and $\overline{r}_{ui}$ is the predicted rating of user u on item i.

This metric is popular as it prefers many small errors over a few large ones (as opposed to the mean absolute error). It was the accuracy metric of the Netflix Challenge [5] and is one of the most widely used error metrics in recommender systems literature today.

1.5.2 Evaluating ranking prediction

We mention four metrics for top K recommendation: *precision*, *recall*, *normalized discounted cumulative gain* (NDCG), and *area under the ROC curve* (AUROC or AUC). Precision, recall and AUROC are traditionally used for evaluating classification problems, while NDCG is used to evaluate ranking lists.

One can categorize these metrics based on whether they differentiate the importance of elements based on their positions in the ranking list. While NDCG considers the prediction errors occurring further down the ranking list with lower weight, precision, recall only consider whether the predicted items are one of the top K elements, and AUROC does the same, but for all possible K values.

Precision and recall

When disregarding the actual ranks of items and treating the ranking list as the set of the top K items and the set of everything else, then the ranking prediction task becomes a special kind of classification problem. For a fixed user u, each item has to be classified either as one of the top K items, or one of the remaining $(m - K)$ items. Precision and recall are defined analogously as in classification:

$$\text{Precision}_u \overset{\text{def}}{=\!=} \frac{\mathit{TruePositive}_u}{\mathit{TruePositive}_u + \mathit{FalsePositive}_u}, \tag{1.8}$$

$$\text{Recall}_u \overset{\text{def}}{=\!=} \frac{\mathit{TruePositive}_u}{\mathit{TruePositive}_u + \mathit{FalseNegative}_u}, \tag{1.9}$$

where $TruePositive_u$ is the number elements correctly predicted to be in the top K, $FalsePositive_u$ is the number of elements incorrectly predicted to be in the top K, $FalsePositive_u$ is the number of elements incorrectly predicted to be in the top K, and $FalseNegative_u$ is the number of elements incorrectly predicted not to be in the top K, as seen in table 1.1.

		Is it predicted to be in the top K	
		yes	**no**
Is it in the top K	**yes**	True Positive	False Negative
	no	False Positive	True Negative

Table 1.1: The confusion matrix for top K recommendation.

In essence, precision is the ratio of items accurately predicted to be positive to all items predicted to be positive, and recall is the ratio of items accurately predicted to be positive to all items that are actually positive.

Normalized discounted cumulative gain

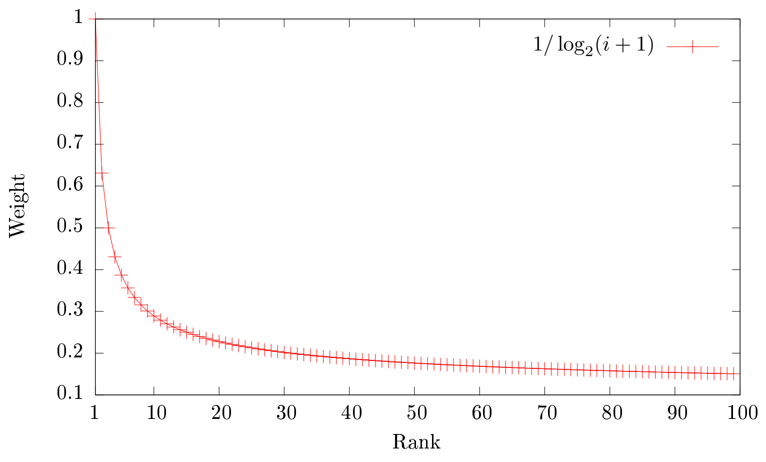


Figure 1.3: The decreasing logarithmic weight function of DCG.

Discounted Cumulative Gain or DCG is a metric that sums the relevance of the items predicted to be in the top K, discounted by a decreasing logarithmic weight function (Figure 3.3). The DCG value of a top K recommendation for user u is given by

$$\mathrm{DCG}\,@K_u \overset{\text{def}}{=} \sum_{i=1}^{K} \frac{\mathrm{rel}(\mathrm{top}_u(i))}{\log_2(i+1)}, \tag{1.10}$$

where $\mathrm{top}_u(i)$ is the i-th item on the predicted ranking list of user u, and $\mathrm{rel}(i)$ is a relevance measure of item i. The relevance used here can be e.g. the observed rating values of the items in the explicit case, or the corresponding $0-1$ value in the implicit case.

Since the value of this expression depends on the the value of K, the definition of relevance and the actual dataset, it is common to normalize it by the maximum achievable DCG value with the given parameters, the *Ideal Discounted Cumulative Gain*. The NDCG value is then given by

$$\mathrm{NDCG}\,@K_u \overset{\text{def}}{=} \frac{\mathrm{DCG}\,@K_u}{\mathrm{IDCG}\,@K_u}. \tag{1.11}$$

The NDCG metric can also be defined without the limit K. In this case, it evaluates the accuracy of the predictions over the total length of the ranking list. We use this version in Section 3.2.

Area under the ROC curve

The area under the *receiver operating characteristic* (ROC) curve is another metric that originates from the area of classifier systems. To define the ROC curve, we need another metric that is similar to precision and recall: the *fall-out*.

$$\text{Fall-out}_u \overset{\text{def}}{=} \frac{\mathit{FalsePositive}_u}{\mathit{FalsePositive}_u + \mathit{TrueNegative}_u}, \tag{1.12}$$

i.e, the fall-out is the ratio of incorrectly rejected items to all negative items.

The ROC curve is defined by plotting the recall on the y axis against the fall-out on the x axis, while changing the threshold of the classifier, in this case, value of K. The likelier the classifier is to accept true positive items first while increasing the threshold, the higher the curve will go above the identity. For this reason, the area under the ROC curve is a good measure for characterizing the accuracy of the classifier at different thresholds at the same time.

This definition of AUROC is equivalent to the following:

$$\mathrm{AUROC} \overset{\text{def}}{=} P\big(\mathrm{rank}\left(M_r^+\right) > \mathrm{rank}\left(M_r^-\right)\big) = \tag{1.13}$$

$$= \frac{1}{|M^+||M^-|} \sum_{i\in M^+} \sum_{j\in M^-} I\big(\,\mathrm{rank}\,(i) > \mathrm{rank}\,(j)\big)\,, \tag{1.14}$$

where $I(.)$ is the indicator function, M^+ is the set of positive items (the items with true rank greater or equal than K), M^- is the set of negative items (the items with true rank less than K), $\mathrm{rank}\,(i)$ is the predicted rank of item i, M_r^+ is a randomly selected positive item and M_r^- is a randomly selected negative item.

The equivalence of the definitions is easily seen: starting with a perfect classifier, when the the predicted rank of all positive items is greater than the predicted rank of all negative items. In this situation, the area under the ROC curve is clearly 1. Now with every inversion we make in the ranking list, the

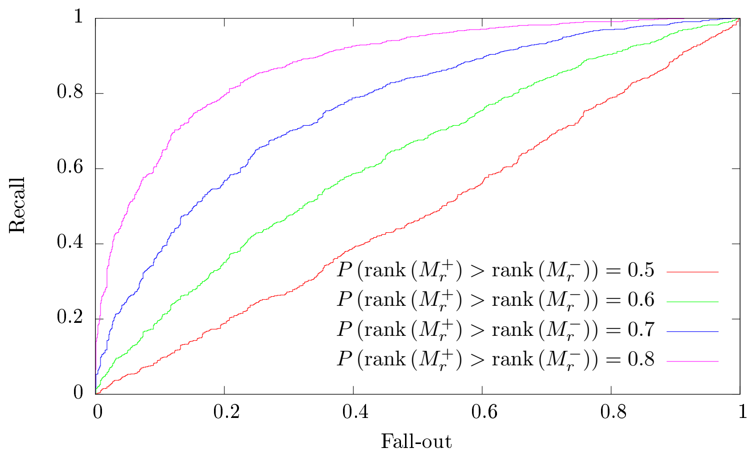


Figure 1.4: ROC curves measuring different performances, with 1000 negative and 1000 positive items, ordered by random scores from appropriate normal distributions. The performance is the probability of a positive item ranking higher than a negative item.

probability $P(\text{rank}(M_r^+) > \text{rank}(M_r^-))$ decreases with $\frac{1}{|M^+||M^-|}$ and so does the area under the ROC curve.

Chapter 2

Matrix factorization

In Section 1.2, we described neighborhood based models for collaborative filtering. Another model for collaborative filtering is the *matrix factorization model.* It has been proved to be very effective, and has become dominant method in the field.

The matrix factorization model works by examining the $n \times m$ matrix R (where n is the number of users and m is the number of items) consisting of the observed r_{ui} values (and missing elements). This matrix is called the *user-item rating matrix.*

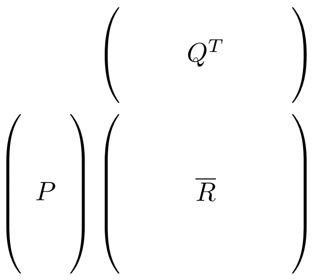


Figure 2.1: Approximating the user-item rating matrix as the product of two low dimension factors, $P \in \mathbb{R}^{n \times f}$ and $Q \in \mathbb{R}^{m \times f}$.

The goal is to find two smaller matrices, $P \in \mathbb{R}^{n \times f}$ and $Q \in \mathbb{R}^{m \times f}$, such that $P \times Q^T = \overline{R}$, where the matrix $\overline{R}$ approximately equals to R according to some objective function. The value f is called *factor dimension*, and is typically orders of magnitude smaller than n and m. Our prediction for a not-yet known r_{ui} value is $\overline{r}_{ui}$, the corresponding element of $\overline{R}$.

In Section 2.1, we summarize the most widely used objective functions and techniques for matrix factorization. In Section 2.2, we describe three algorithms for factoring the user-item rating matrix. Finally, in Section 2.3, we describe a slightly different variant of the matrix factorization model.

2.1 Objective functions

The objective function that the a matrix factorization algorithm minimizes is a real valued function defined on the state of the recommendation model and the training data. In the case of the matrix factorization model, it usually computes the accuracy of the model using the elements of the product matrix and their observed counterparts.

A naturally occurring idea is to use the evaluation functions as objective functions. However, when training the model, this is not always the best choice. Models can be prone to overfitting, the training data is not always reliable and unbiased, and as we show in Section 3 and Section 4, sometimes the evaluation function's properties are simply not ideal for training the model.

2.1.1 Pointwise, pairwise and listwise

The RMSE objective function works by measuring differences between ratings and their predicted counterparts, and is a classic example of a pointwise objective function.However, as described in Section 1.1, ranking prediction doesn't require the predicted ratings to be close to particular values, because only their relative sizes matter when computing the predicted ranking list. Pairwise objective functions consider pairs of ratings and predicted ratings, and train the model based on their relative size differences. They can be derived from a Bayesian analysis of the ranking problem [1], or equivalently, by optimizing for AUROC and replacing the non-differentiable indicator function with a differentiable function [6]. Listwise is another approach, where an individual training example is an entire list of items, rather than indiviual items or item pairs [7].

2.1.2 Overfitting and regularization

The notion of overfitting is a well known phenomenon in machine learning. When a model's predictions fit very closely to the training data, it inevitably learns the random noise in the data among the underlying relationships. This makes it so that while the model seems to achieve better performance on the training data, it performs poorly on the test dataset. There are multiple techniques for preventing overfitting, including for example the usage of a validation set (see Section 1.3) to detect overfitting or the usage of regularization terms.

Regularization terms are parts of the objective function that are included with the purpose of penalizing the model fitting too accurately to the training data, or to enforce some kind of desired property of the model. The most common method of regularization when using matrix factorization is to include the Euclidean norm of the rows of the factors, P and Q. The most common pointwise objective function to be minimized with matrix factorization models is the squared error with this term added, i.e

$$\min_{P,Q} \sum_{(u,i)\in H} \left(r_{ui} - p_u q_i^T\right)^2 + \lambda \left(\|p_u\|^2 + \|q_u\|^2\right), \tag{2.1}$$

where H denotes the training set, p_u is the row of P corresponding to user u, and q_i is the row of Q corresponding to item i. This regularization term is called L^2 regularization.

In the case of implicit recommendation, an important thing to consider is that since (as described in Section 1.4) negative feedback is inherently nonexistent, the model has to be supplied with artificially generated negative feedback. One could attempt to consider all missing data as negative feedback by assigning zero to the corresponding elements of the user-item rating matrix. However this approach has a huge problem: since the matrix is sparse, this would make it so that predicting constant zero for every user-item pair is a fairly good model according to RMSE. Even when using a model with enough expressiveness to overcome this problem, the user-item pairs for which the system must predict the ratings would tend to be zero valued.

One solution to this problem is to randomly select a subset of the possible user-item pairs and assign zero only to corresponding elements of the user-item matrix. This process is called negative sample generation and is usually the preferred way of solving the problem when using stochastic gradient descent (see Section 2.2.3). Another solution is to consider all missing data zero valued, but with less weight in the objective function. We describe an example of this approach in Section 2.2.2.

2.2 Algorithms for matrix factorization

In this Section, we present three algorithms used for factoring the user-item matrix as the product of two low-dimension matrices.

2.2.1 Singular value decomposition

Singular value decomposition is a well known way to decompose a matrix. The statement of the theorem, as described in [8], is the following:

Theorem 2.2.1. *For any real matrix α, two orthogonal matrices u and U can be found so that*

$$\lambda = u\alpha U \tag{2.2}$$

is a real diagonal matrix with no negative elements.

The equation in the theorem is equivalent to the statement that any $\alpha \in \mathbb{R}^{m\times n}$ matrix can be decomposed as

$$\alpha = u\lambda U, \tag{2.3}$$

where $\lambda \in \mathbb{R}^{n\times m}$ is a non-negative diagonal matrix, and $u \in \mathbb{R}^{n\times n}, U \in \mathbb{R}^{m\times m}$ are orthogonal matrices. The elements of λ are the singular values of the matrix α. Note that the order of elements in λ can be chosen arbitrarily, when changing u and U accordingly. Here, as per convention, we assume they are listed in descending order. If $n < m$, the matrix λ has $m - n$ columns of zeros, thus the value of α only depends on the first n rows of U.

This theorem has multiple applications. The one relevant here is known as the Eckart-Young theorem which states the following:

Theorem 2.2.2 (Eckart-Young)**.** *When approximating the matrix $\alpha \in \mathbb{R}^{n\times m}$ with matrix $\overline{\alpha} \in \mathbb{R}^{n\times m}$ such that* $\operatorname{rank}(\overline{\alpha}) \leq r$*, the matrix that minimizes $\sum_{i=1}^{n}\sum_{j=1}^{m}(\alpha_{ij} - \overline{\alpha}_{ij})^2$ (i.e the Frobenius norm of the difference) is given by*

$$\overline{\alpha} = \overline{u}\overline{\lambda}\overline{U} \tag{2.4}$$

where $\overline{\lambda} \in \mathbb{R}^{r\times r}$ is the diagonal matrix consisting of the r largest singular values of α (i.e λ truncated to the first r rows and columns), $\overline{u} \in \mathbb{R}^{n\times r}$ is u truncated to the first r columns, and $\overline{U} \in \mathbb{R}^{r\times m}$ is U truncated to the first r rows.

Knowing these theorems, it is a naturally occurring idea that the SVD along with the Eckart-Young theorem can be used to decompose the user-item rating matrix R into the two factors, $P = \overline{u}$ and $Q = (\overline{\lambda}\overline{U})^T$. However, since R is usually very sparse, and the method described above minimizes for least squares considering *all* the elements of the matrices, the unknown elements of R have to be filled prior to decomposing the matrix. Sarwar et al. [9] suggest that using the item-average ratings (the average of ratings that the item in question has) for this purpose yields good results.

While SVD guarantees the best least squares approximation of the user-item matrix, this approach has multiple problems. The first problem is that this method fills in the missing elements of R, making it a dense matrix. Since the decomposition algorithm requires $\mathcal{O}(m^3)$ time [10], this makes this approach computationally inefficient, especially when the recommender system is expected to function in an online fashion. To make the algorithm feasible in an online setting, techniques for incrementally building SVD-based models have been proposed [10].

The second problem is that since SVD produces the best least squares approximation possible, it is prone to overfitting [1], see Section 2.1.2. Utilizing regularization is not trivial, since the theorem only holds for least squares optimization (with equal weights), i.e the Frobenius norm.

For these reasons, generally SVD is usually a better fit to be used in conjunction with kNN, described in Section 1.2. The process of approximating R via a lower rank matrix can be thought of as a dimensionality reduction process that filters unnecessary noise, and the similarity measures can be defined based on the factors produced by SVD, instead of columns and rows of R.

2.2.2 Alternating least squares

The alternating least squares (ALS) algorithm factorizes the user-item rating matrix R while minimizing most commonly the following objective function (as seen in Section 2.1.2):

$$\min_{P,Q} \sum_{(u,i)\in H} \left(r_{ui} - p_u q_i^T\right)^2 + \lambda\left(\|p_u\|^2 + \|q_u\|^2\right). \tag{2.5}$$

The alternating least squares algorithm works by taking turns fixing P and Q and optimizing the other for the above objective function. With this, the value of the objective function decreases monotonically. This process is done based on the technique called *ridge regression* [11].

Ridge regression takes a matrix $X = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^{n\times f}$ and target values $y = (y_1, y_2, \ldots, y_n) \in \mathbb{R}^n$ and finds $w \in R^f$ in such a way that $x_i w$ approximates y_i while also keeping the value $w^T w$ small. More accurately, ridge regression seeks to minimize the following error function as a function of w:

$$err = \lambda w^T w + \sum_{i=1}^{n} (y_i - x_i w)^2 = \lambda\|w\|^2 + \|y - Xw\|^2, \tag{2.6}$$

where $\lambda \in \mathbb{R}$ is a given regularization coefficient. By derivation, we can find the optimum value for w analytically[1]: let w be such that

$$\frac{\partial err}{\partial w} = 2\lambda w + 2(X^T X w - X^T y) = 0, \tag{2.7}$$

which is equivalent to

$$\begin{aligned} \lambda w + X^T X w - X^T y &= 0, & (2.8)\\ \left(\lambda I + X^T X\right) w &= X^T y, & (2.9)\\ w &= \left(\lambda I + X^T X\right)^{-1} X^T y. & (2.10) \end{aligned}$$

Note that the matrix being inverted is of size $f \times f$.

This method can be applied for alternating least squares calculation: for example, when fixing Q and optimizing for P, each p_u row of P along with the observed r_{ui} ratings of user u and the corresponding q_i rows from Q form an optimization problem that can be solved with ridge regression. This makes it so that optimizing for P takes $\mathcal{O}\big(\sum_{u=1}^{n}\left(m_u f^2 + f^3\right)\big) = \mathcal{O}\big(\mathcal{N} f^2 + n f^3\big)$ time, where f is the factor dimension, n is the number of users, m_u is the number of items rated by user u, and $\mathcal{N} = \sum_{u=1}^{n} m_u$ is the number of the known elements of the user-item rating matrix. Optimizing for Q while fixing P is done similarly.

Applying ALS for online recommendation is done by processing the input data in batches, instead of incrementally updating the model. Even though this makes it less ideal for this task, experimental

[1]The formula used here for calculating the derivative is $\frac{\partial\|b-Ax\|^2}{\partial x} = 2\left(A^T A x - A^T b\right)$

results prove that it is worth considering, since the accuracy of the predictions of the model is relatively good.

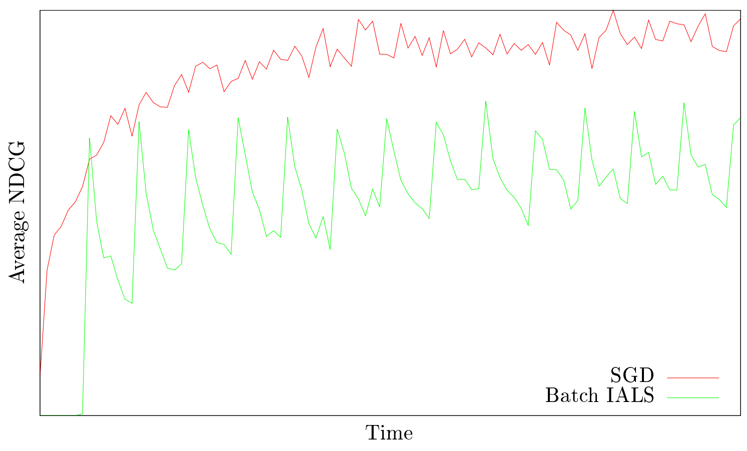


Figure 2.2: Batch processing for online recommendation (IALS) vs online recommendation with stochastic gradient descent.

ALS for implicit recommendation

When applying ALS for implicit feedback data, there is an additional difficulty to consider. In the explicit case it is possible to receive possible and negative feedback as well. In the implicit case however, the only kind of feedback is positive, so we have to artificially generate negative feedback (see Section 2.1.2). In practice, this either means randomly including unknown elements in the optimization process and assigning them the value 0, or considering all missing items zero valued. If the latter is used, this means that the user-item rating matrix is now a dense matrix, thus we have to consider *all* user-item pairs known. Since the runtime of ALS is a function of number of known elements of the user-item rating matrix, the algorithm described above may become infeasible for implicit recommendation.

In their 2008 paper [12], Koren et. al. present a slightly modified model for the implicit setting. Instead of discarding the number of times a user has interacted with an item, they introduce the notion of confidence. The authors suggest that while the numerical value of explicit feedback indicates preference, the numerical value of implicit feedback (for example the number of times a user has listened to a song) indicates confidence. It is thus necessary to separate implicit feedback into two separate variables, s_{ui} and c_{ui}, where s_{ui} is a binary variable indicating the relation between the user and the item (positive or negative) and c_{ui} indicates our confidence in the value of s_{ui}. The authors suggest to have $s_{ui} = 1$ if $r_{ui} \geq 1$ and 0 otherwise, and that c_{ui} be an increasing function of r_{ui}, for example $c_{ui} = 1 + \alpha \cdot r_{ui}$. Through experimentation they found $\alpha = 40$ to be a suitable value, however the optimum heavily depends on the actual dataset.

The objective function of the optimization process is then defined as

$$err = \sum_{u,i} c_{ui} \left(s_{ui} - p_u q_i^T\right)^2 + \lambda \left(\sum_u \|p_u\|^2 + \sum_i \|q_i\|^2\right), \tag{2.11}$$

which can also be formulated as

$$err = \left(\sum_u \left\|\sqrt{C^u} s(u) - \sqrt{C^u} Q p_u\right\|^2\right) + \lambda \left(\sum_u \|p_u\|^2 + \sum_i \|q_i\|^2\right), \tag{2.12}$$

where $s(u) \in \mathbb{R}^m$ is the vector of s_{ui} preferences, $C^u \in \mathbb{R}^{m \times m}$ is a diagonal matrix consisting of the c_{ui} confidence values, and $\sqrt{C^u} \in \mathbb{R}^{m \times m}$ is the diagonal matrix consisting of square roots of the c_{ui} confidence values.

Through differentiation by a particular row of P, we can obtain the optimum of the objective function analytically: let p_u be such that

$$\frac{\partial}{\partial p_u} err = 2\left(Q^T C^u Q p_u - Q^T C^u s(u)\right) + 2\lambda p(u) = 0, \tag{2.13}$$

which is equivalent to

$$p_u = (Q^T C^u Q + \lambda I)^{-1} Q^T C^u s(u). \tag{2.14}$$

We can find the optimum for the q_u values similarly.

Expression 2.14 has to be calculated for every p_u and q_u vector in an alternating fashion. A significant speedup can be achieved through the realization that

$$\left(Q^T C^u Q + \lambda I\right)^{-1} Q^T C^u s(u) = \tag{2.15}$$

$$= \left(\left(Q^T Q + Q^T \left(C^u - I\right) Q\right) + \lambda I\right)^{-1} Q^T C^u s(u). \tag{2.16}$$

Here the value $Q^T Q$ has to be calculated only once per iteration, and both $(C^u - I)$ and $s(u)$ consist of only m_u nonzero elements, where m_u is the number of items that the user has interacted with. Note that similar to what we've seen in the explicit case, the matrix being inverted is of size $f \times f$.

One iteration of this algorithm can thus be computed in $\mathcal{O}\left(f^2 \mathcal{N} + n f^3\right)$ time, which makes it linear in the size of the input, $\mathcal{N}$.

2.2.3 Stochastic gradient descent

Gradient descent is an optimization algorithm to find a (local) minimum of a differentiable multi-variable function. It works by starting from any point in the parameter space and iteratively taking steps in the direction of the steepest descent, i.e the negative of the gradient.

Stochastic gradient descent (SGD) is a stochastic estimation of the gradient descent algorithm . It deals with a special case of above problem where the objective function to be minimized is the sum of multiple differentiable functions:

$$obj(w) = \sum_i F_i(w). \tag{2.17}$$

It achieves computational efficiency by approximating the gradient in each step of the algorithm. The gradient of 2.17 by w takes the following form:

$$\frac{\partial obj(w)}{\partial w} = \sum_i \frac{\partial F_i(w)}{\partial w}. \tag{2.18}$$

Calculating the right side of this equation can be computationally expensive, for example when the sum ranges over the possibly enormous training set of a machine learning algorithm. One step of SGD approximates the gradient of the objective function by a single term of the above sum, while iterating over the possible values of i:

$$\forall i: \quad w \leftarrow w - \gamma \nabla F_i(w). \tag{2.19}$$

The coefficient γ is called the *learning rate*.

SGD is one of the most popular algorithms for recommendation via matrix factorization. It generally liked because it has multiple advantageous properties: it can minimize any differentiable objective function in the form of Equation 2.17; it can update the model iteratively when additional observations are obtained; and it can be manipulated to treat parts of the training data as more important than others by changing the learning rate, the frequency and the order of the (u, i) pair processed.

We can differentiate between *Offline* (or *batch*) SGD and *Online* SGD. Optimizing for the objective function

$$err = \sum_{(u,i)} \left(r_{ui} - p_u q_i^T\right)^2 + \lambda \left(\sum_u \|p_u\|^2 + \sum_i \|q_i\|^2 \right), \tag{2.20}$$

yields the with gradients

$$\frac{\partial err}{\partial p_u} = -2 \left(\sum_i \left(r_{ui} - p_u q_i^T\right) q_i \right) + 2\lambda p_u \text{ and} \tag{2.21}$$

$$\frac{\partial err}{\partial q_i} = -2 \left(\sum_u \left(r_{ui} - p_u q_i^T\right) p_u \right) + 2\lambda q_i. \tag{2.22}$$

Offline SGD works by iterating over all possible (u, i) pairs in random order, and updates the corresponding p_u and q_i vectors:

$$\forall (u, i) \in \sigma(N \times M) : \tag{2.23}$$

$$p_u \leftarrow p_u + \gamma((r_{ui} - p_u q_i^T) q_i - \lambda p_u) \tag{2.24}$$

$$q_i \leftarrow q_i + \gamma((r_{ui} - p_u q_i^T) p_u - \lambda q_i) \tag{2.25}$$

where $\sigma \in S_{nm}$ is a random permutation of the user-item pairs.

Online SGD on the other hand iterates over the training data in time wise order, and performs the calculations described in 3.34 a 3.35 for each (u, i, r_{ui}) training record once.

In the implicit case, this version of online SGD would not yield any useful results, as the explicitly known values of the matrix R are all valued 1. The usual solution to this problem is to generate random unknown (u, i) pairs and run an iteration of the above algorithm with $r_{ui} = 0$. This technique is called *negative sample generation* (see Section 2.1.2). The ratio of positive to negative samples is called *negative rate*, and it can be chosen freely to suit the particular optimization problem.

2.3 Asymmetric matrix factorization

Asymmetric matrix factorization is a slightly different matrix factorization model, proposed by [13] for implicit recommendation via stochastic gradient descent. The main idea behind asymmetric matrix factorization is that instead of the user factor $P \in \mathbb{R}^{n \times f}$, we define a second $Z \in \mathbb{R}^{m \times f}$ matrix and replacing p_u values with virtual p_u^* values defined as

$$p_u^* \overset{\text{def}}{=\!=} \frac{1}{\sqrt{|H_u|}} \sum_{i \in H_u} z_i, \tag{2.26}$$

where z_i denotes the row of Z corresponding to item i and $H_u = \{i \mid r_{ui} = 1\}$.

Our prediction for an unknown element of the user-item rating matrix is then calculated as

$$\overline{r}_{ui} = p_u^* q_i^T = \frac{1}{\sqrt{|H_u|}} \sum_{j \in H_u} z_j q_i^T. \tag{2.27}$$

The matrix factorization process works similarly to the one described in Section 2.2.3, with virtual p_u^* vectors instead of p_u vectors. Note however, that when updating the asymmetric matrix model, we can't directly modify the values of p_u^* vectors, instead we have to update the corresponding z_i vectors. When intending to increase the value of p_u^* by the value of an arbitrary vector $\alpha \in \mathbb{R}^f$, i.e

$$p_u^* \leftarrow p_u^* + \alpha, \tag{2.28}$$

we have the following equation according to the definitions:

$$p_u^* + \alpha = \frac{1}{\sqrt{|H_u|}} \sum_{i \in H_u} \left(z_i + \frac{\alpha}{\sqrt{|H_u|}} \right), \tag{2.29}$$

thus we have to perform the following calculations:

$$\forall j \in H_u : \ z_j \leftarrow z_j + \frac{\alpha}{\sqrt{|H_u|}}. \tag{2.30}$$

The SGD algorithm is modified accordingly, updating the corresponding z_j vectors instead of p_u. When optimizing for MSE with regularization, this means that one iteration with a given (u, i, r_{ui}) record is calculated as follows:

$$\forall j \in H_u : \ z_j \leftarrow z_j + \frac{\gamma}{\sqrt{|H_u|}}((r_{ui} - p_u^* q_i^T) q_i - \lambda p_u^*) \tag{2.31}$$

$$q_i \leftarrow q_i + \gamma((r_{ui} - p_u^* q_i^T) p_u^* - \lambda q_i) \tag{2.32}$$

One advantage of this approach is that when a new user enters the system, we can immediately produce recommendations based on the preferences of the user, with very little performance cost. This makes it ideal for online recommendation.

Chapter 3

Ranking based objective functions

The most popular objective function for matrix factorization is the root mean square error. Experiments show that optimizing for RMSE yields great results not only for rating prediction, but for ranking prediction as well. It is generally accepted that good performance according to RMSE correlates with good performance measured in NDCG, or other ranking based evaluation functions. Many algorithms have been developed specially for minimizing RMSE. However, as described in Section 2.2.3, SGD is only able to utilize differentiable error functions.

It is tempting to attempt to use the evaluation function directly for optimization. This, however, is not possible with ranking based evaluation functions such as NDCG or AUROC, as they are not differentiable with respect to the predicted $\overline{r_{ui}}$ ratings. In their 2015 paper [14] *Gaussian Ranking by Matrix Factorization*, Steck et al. introduced a general approach for optimizing toward ranking metrics, based on the assumption that the predicted ratings for a given user follow approximately a Gaussian distribution.

In Section 3.1, we describe the ideas presented in [14]. In Section 3.2, we summarize the application of these ideas for the special cases of NDCG and AUROC in the implicit setting. Finally, in Section 3.3 and 3.4 we present our own approach to applying this method for implicit online recommendation.

3.1 Gaussian ranking

Optimizing directly for ranking metrics using SGD is not possible, since the metric is a function of the positions of the items in the ranking list, which is not differentiable with respect to the predicted ratings of the items. However, in many cases, the evaluation metric is differentiable with respect to the ranks of the items. This is the case for example with NDCG and AUROC.

If we approximate the ranks of the items with a differentiable function of their predicted ratings, then we can differentiate the above mentioned evaluation metrics with respect to the predicted ratings of the items. This makes it possible to use these functions as objective functions for the learning process. In the remainder of this thesis, we adopt the notation of [14]: we will call the predicted ratings of the items *scores* or predicted scores and use the notation s_{ui} for rating of user u on item i. We will also use $rank^*_{ui}$ for the actual rank of item i on the ranking list of user u, and $rank_{ui}$ for the predicted rank of item i on the ranking list of user u.

Steck et al. suggest approximating the ranks of the items based on the assumption that the predicted scores of the items follow approximately a Gaussian distribution. Since the scores are predicted by summing over the f latent dimensions of the factors P and Q, based on the central limit theorem this assumption is reasonable.

They define the predicted ranks of the items via the formula

$$rank_{ui}(s_{ui}^{(n)}) \stackrel{\text{def}}{=} \max\left\{1, m\left(1 - \Phi\left(s_{ui}^{(n)}\right)\right)\right\}, \tag{3.1}$$

where Φ is the cumulative distribution function of the standard normal distribution. We define the standardized score $s^{(n)}$ as

$$s_{ui}^{(n)} \stackrel{\text{def}}{=} \frac{s_{ui} - \mu_u}{\sigma_u}, \tag{3.2}$$

where μ_u and σ_u denote the expected value and variance of the distribution of scores for the particular user, u.

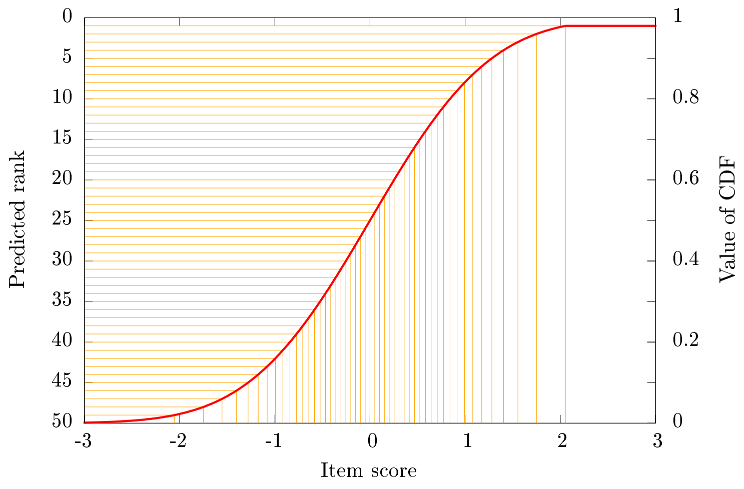


Figure 3.1: The predicted ranks based on the score with 50 items.

Given a ranking metric in the form $\sum_{ui} L$, with the loss function L being differentiable with respect to the ranks of items, using the above approximation it is now possible to differentiate L with respect to a model parameter θ, by applying the chain rule [14]:

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial rank} \frac{\partial rank}{\partial s^{(n)}} \frac{\partial s^{(n)}}{\partial s} \frac{\partial s}{\partial \theta}. \tag{3.3}$$

The coefficients $\frac{\partial rank}{\partial s^{(n)}}$ and $\frac{\partial s^{(n)}}{\partial s}$ can be calculated based on definitions 3.1 and 3.2:

$$\frac{\partial rank_{ui}}{\partial s_{ui}^{(n)}} = -N\phi\left(s_{ui}^{(n)}\right), \tag{3.4}$$

$$\frac{\partial s_{ui}^{(n)}}{\partial s_{ui}} = \frac{1}{\sigma_u}, \tag{3.5}$$

where ϕ is the probability density function of the standard normal distribution. Note, that when calculating the derivative of $rank$ by $s^{(n)}$, we ignore the maximum operator from the definition in Equation 3.1. We will discuss examples of calculating $\frac{\partial L}{\partial rank}$ and $\frac{\partial s}{\partial \theta}$ in Section 3.2.

As seen on Figure 3.2, while the Gaussian assumption seems to hold at the start of the optimization process, the distribution of scores does not appear to be Gaussian when training the model via minimizing RMSE. Indeed, Steck et al. propose adding the regularization term $\alpha \sum s_{ui}^2$ to the objective function to enforce normal distribution, where α is a properly chosen weight coefficient. With this, the objective

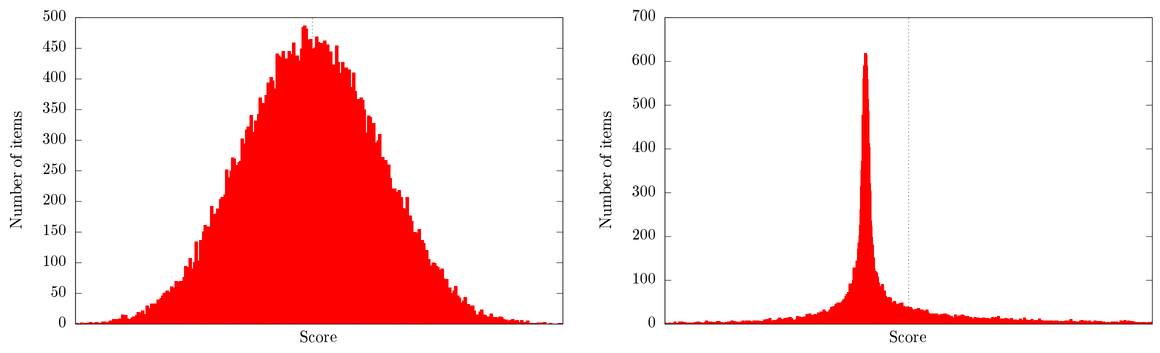


Figure 3.2: On the left: empirical distribution of scores at the start of the learning process, with a sample of 50000 items. On the right: empirical distribution of scores after learning with SGD with RMSE as objective function, standardized with the sample mean and variance.

function to be minimized for matrix factorization becomes

$$\sum_{u,i} \left(L(s_{ui}) + \lambda \left(\|p_u\|^2 + \|q_i\|^2\right)\right) + \alpha \sum_{u,i} s_{ui}^2. \tag{3.6}$$

3.2 Differentiating NDCG and AUROC

In Section 3.1, we have introduced a method for differentiating ranking based metrics with respect to the items scores. In this Section, we summarize the application of this method for differentiating NDCG and AUROC with respect to the item scores in the implicit setting, as described in [14]. Since these metrics are defined *per user*, we omit the indices indicating the user in the following formulas.

The NDCG ranking metric is defined as $\frac{\text{DCG}}{\text{IDCG}}$, while DCG is defined as $\sum_{i=1}^{K} \frac{\text{rel}(\text{top}_u(i))}{\log_2(i+1)}$. It is more beneficial, however, to use the version where the sum ranges over the total length of the ranking list (see Section 1.5.2). When dealing with implicit recommendation, the number of interaction between users and items is usually discarded, meaning the relevance of the items is either 1 or 0 valued. This reduces the DCG metric to

$$\text{DCG} = \sum_{i \in M^+} \frac{1}{\log_2(rank_i + 1)}, \tag{3.7}$$

where M^+ is the set of positive items. To obtain a loss function, we have to negate the NDCG metric. Also note that Equation 3.7 only depends on the ranks of the positive items, thus we are not able to differentiate the formula by the ranks of negative items. To avoid confusion, we will indicate that a given $rank_i$ rating is the rating of a positive item by writing $rank_i^+$.

With these considerations, we can calculate the derivative of the NDCG metric as a loss function as follows:

$$\frac{\partial L}{\partial rank_i^+} = \frac{\partial(-\text{NDCG})}{\partial rank_i^+} = \frac{1}{\text{IDCG}} \cdot \frac{1}{\left(rank_i^+ + 1\right) \log_2^2\left(rank_i^+ + 1\right)}. \tag{3.8}$$

Note, that the ranks defined by Equation 3.1 are independent variables, unlike the real ranks of items. This makes it possible to differentiate L term-by-term in Equation 3.8.

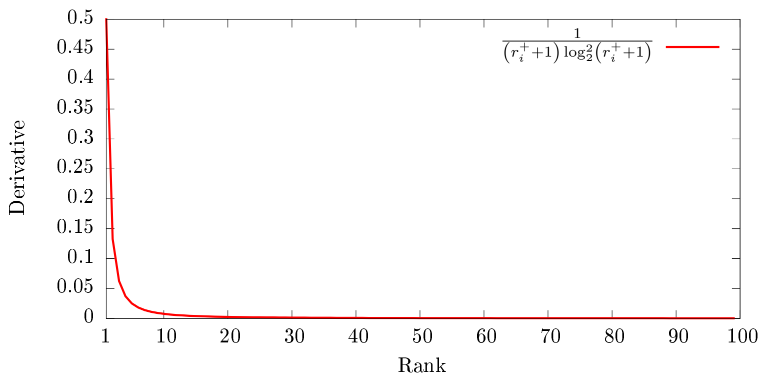


Figure 3.3: A single term of the derivative of the approximated DCG function in Equation 3.8.

AUROC is defined as

$$\text{AUROC} \overset{\text{def}}{=} \frac{1}{m^+ m^-} \sum_{i \in M^+} \sum_{j \in M^-} I(\text{rank}\,(i) > \text{rank}\,(j))\,, \tag{3.9}$$

where M^+ is the set of positive items, M^- is the set of negative items, $m^+ = |M^+|$ and $m^- = |M^-|$. As described in [14], this formula can be expressed in the alternative form

$$\text{AUROC} = \frac{1}{m^+ m^-} \left((m+1)m^+ - \binom{m^+ + 1}{2} - \sum_{i \in M^+} rank_i^+ \right). \tag{3.10}$$

The equality of the two expressions can be seen the following way: since $m = m^+ + m^-$, we can simplify the sum on the right side as

$$(m+1)m^+ - \binom{m^+ + 1}{2} - \sum_{i \in M^+} rank_i^+ = \tag{3.11}$$

$$= \left(m^+ m^- + m^+ m^+ + m^+\right) - \frac{m^+ m^+ + m^+}{2} - \sum_{i \in M^+} rank_i^+ = \tag{3.12}$$

$$= m^+ m^- + \frac{m^+ m^+ + m^+}{2} - \sum_{i \in M^+} rank_i^+ = \tag{3.13}$$

$$= m^+ m^- + \binom{m^+ + 1}{2} - \sum_{i \in M^+} rank_i^+. \tag{3.14}$$

Now consider the equation

$$\sum_{i \in M^+} \sum_{j \in M^-} I(\text{rank}\,(i) > \text{rank}\,(j)) = m^+ m^- + \binom{m^+ + 1}{2} - \sum_{i \in M^+} rank_i^+. \tag{3.15}$$

Assuming a perfect recommendation, i.e the case when $\{rank_i^+ \mid i \in M^+\} = \{1, 2, \ldots, m^+\}$, the two sides of Equation 3.15 are clearly equal. Now (similarly as in Section 1.5.2), with every inversion in the ranking list, both sides of the equation decrease by one. This proves Equation 3.10.

The equivalent formula in Equation 3.10 shows that AUROC as a loss function, i.e the negative of AUROC depends linearly on the ranks of the items. This means that the derivative of the metric with respect to the rank of a positive item is constant:

$$\frac{\partial L}{\partial rank_i^+} = \frac{\partial(-\,\text{AUROC})}{\partial rank_i^+} = \frac{1}{m^+ m^-}. \tag{3.16}$$

As we can see, both derivatives in Equation 3.8 and Equation 3.16 only depend on the ranks of the positive items, thus one iteration of the SGD algorithm can only be run on positive samples. Steck et al. in [14] propose learning on negative samples via utilizing the regularization term introduced in Equation 3.6. This also helps balance the learning process. Note, however, that there is no situation where the scores are pulled below zero. The authors suggest that this is not a problem, since typical real world applications (and also NDCG@K) only care about the head of the ranking list, i.e the items with the largest (positive) scores.

3.3 Ranking based objective functions for online recommendation

In this section, we describe our own approach for applying the ideas described in Sections 3.1 and 3.2 in an online fashion. Reportedly, the objective functions described in 3.2 (and their listwise variants, see [14]) have produced great results when applied for offline implicit recommendation in combination with AMF (see Section 2.3). Our research was aimed at trying to adopt them with similar performance for implicit online recommendation.

With online recommendation, it is very important for the model to be able to quickly react to changes in user preferences. For this reason, it is beneficial to be able to learn not only on positive samples, but negatives samples as well. We have considered the possibility of formulating NDCG and AUROC in such a way that they depend on the positive and negatives samples as well, and concluded that it is indeed possible.

NDCG

For NDCG, the following equation holds:

$$\text{DCG} = \sum_{i\in M^+} \frac{1}{\log_2(rank_i^+ + 1)} = \sum_{i=1}^{m} \frac{1}{\log_2(i+1)} - \sum_{i\in M^-} \frac{1}{\log_2(rank_i^- + 1)}, \tag{3.17}$$

which makes it possible to write NDCG as the convex combination of the two forms, i.e

$$\text{DCG} = \alpha \left(\sum_{i\in M^+} \frac{1}{\log_2(rank_i^+ + 1)} \right) + \beta \left(\sum_{i=1}^{m} \frac{1}{\log_2(i+1)} - \sum_{i\in M^-} \frac{1}{\log_2(rank_i^- + 1)} \right), \tag{3.18}$$

where α and β are nonnegative real coefficients such that $\alpha + \beta = 1$. We can then use the method described in Sections 3.1 and 3.2 to differentiate NDCG by both positive and negative samples. The derivative of NDCG with respect to the ranking of a positive and a negative item can be calculated as

$$\frac{\partial(-\,\text{NDCG})}{\partial rank_i^+} = \frac{\alpha}{\text{IDCG}\left(rank_i^+ + 1\right)\log_2^2\left(rank_i^+ + 1\right)}, \tag{3.19}$$

$$\frac{\partial(-\,\text{NDCG})}{\partial rank_i^-} = -\frac{\beta}{\text{IDCG}\left(rank_i^- + 1\right)\log_2^2\left(rank_i^- + 1\right)}, \tag{3.20}$$

which means that the derivative of the loss function with respect to a negative sample is the same as the negative of the derivative with respect to a positive sample.

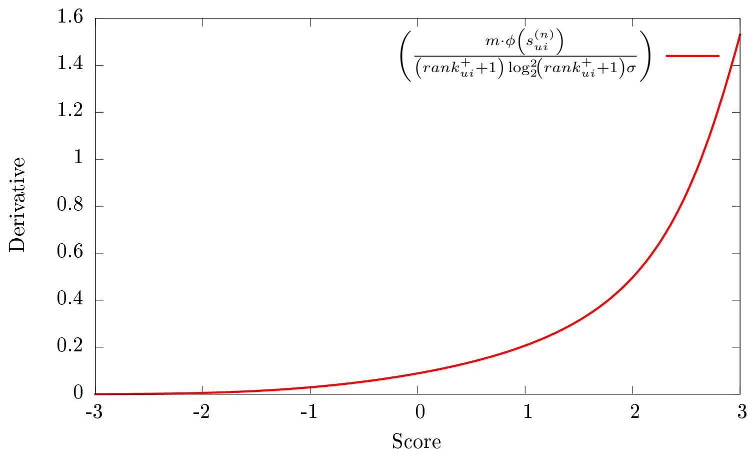


Figure 3.4: The length of the update vector used in SGD with NDCG as objective function, assuming $m = 1000$, $\sigma = 1$ and $\mu = 0$.

With this, the only coefficient missing from Equation 3.3 is $\frac{\partial s_i}{\partial \theta}$. When using SGD, the parameter θ is either p_u or q_i. Since $\frac{\partial s_{ui}}{\partial p_u} = q_i$ and $\frac{\partial s_{ui}}{\partial q_i} = p_u$, using Equations 3.4, 3.5, and 3.8, the gradient of the objective function can be written as

$$\frac{\partial\left(-\,\mathrm{NDCG}\right)}{\partial p_u} \propto \sum_i \left(\frac{\mathrm{sgn}_{ui}\cdot m\cdot\phi\left(s_{ui}^{(n)}\right)}{\left(rank_{ui}^{+}+1\right)\log_2^2\left(rank_{ui}^{+}+1\right)\sigma}\right) q_i, \tag{3.21}$$

where $\mathrm{sgn}_{ui} = -1$ if $i \in M^+$ and $\mathrm{sgn}_{ui} = 1$ if $i \in M^-$. The gradient with respect to q_i can be calculated similarly. In the actual SGD algorithm, only one term of the above sum is used in one update. Also note, that the update happens in the *opposite* direction of the gradient.

AUROC

For AUROC, a similar calculation can be done. Since

$$\sum_{i\in M^+} rank_i^{+} = \binom{m+1}{2} - \sum_{i\in M^-} rank_i^{-}, \tag{3.22}$$

the value of AUROC can be expressed as

$$\mathrm{AUROC} = m^+m^- + \binom{m^+ +1}{2} - \sum_{i\in M^+} rank_i^{+} = \tag{3.23}$$

$$= m^+m^- + \binom{m^+ +1}{2} - \binom{m+1}{2} + \sum_{i\in M^-} rank_i^{-}. \tag{3.24}$$

Taking the convex combination of the two forms similarly to Equation 3.18:

$$\mathrm{AUROC} = m^+m^- + \binom{m^+ +1}{2} - \alpha\left(\sum_{i\in M^+} rank_i^{+}\right) + \beta\left(\sum_{i\in M^-} rank_i^{-} - \binom{m+1}{2}\right), \tag{3.25}$$

the derivative with respect to the ranking of a positive and a negative item can be calculated as

$$\frac{\partial(-\text{AUROC})}{\partial rank_i^+} = \frac{\alpha}{m^+m^-}, \tag{3.26}$$

$$\frac{\partial(-\text{AUROC})}{\partial rank_i^-} = -\frac{\beta}{m^+m^-}, \tag{3.27}$$

meaning that similar to NDCG, the derivatives only differ in sign.

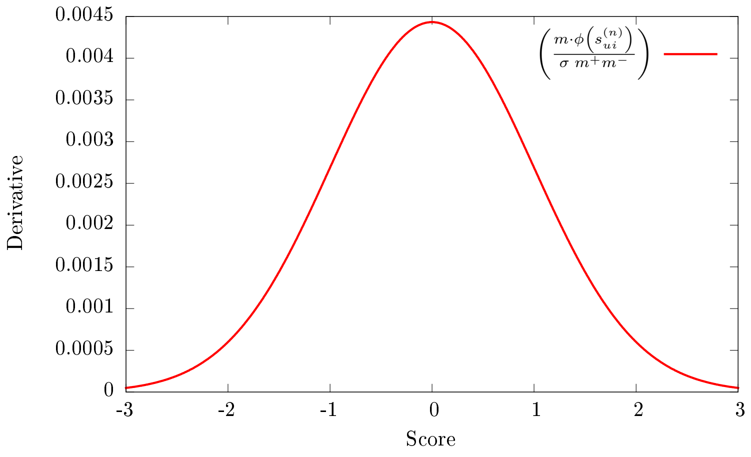


Figure 3.5: The length of the update vector used in SGD with AUROC as objective function, assuming $m = 1000$, $m^+ = 100$, $m^- = 900$, $\sigma = 1$ and $\mu = 0$.

With this, the full expression for the gradient to be used in SGD is

$$\frac{\partial(-\text{AUROC})}{\partial p_u} \propto \sum_i \left(\frac{\text{sgn}_{ui}\, m \cdot \phi\left(s_{ui}^{(n)}\right)}{\sigma\ m^+m^-}\right) q_i. \tag{3.28}$$

Mean squared rank error

Another objective function we examined is what we call *mean squared rank error*. This function is analogous to mean squared error, with the difference that it measures the error of predictions based on the ranks of the items, instead of the scores of the items. It can be defined as

$$\text{MSRE}_u = \frac{1}{m}\sum_{i\in M}\left(\frac{rank_{ui} - rank_{ui}^*}{m}\right)^2. \tag{3.29}$$

where $rank_{ui}^*$ denotes the real rank of item i on the ranking list of user u. The (inner) $\frac{1}{m}$ coefficient is necessary because otherwise the metric would heavily depend on the amount of items in the system, i.e the the possible maximum value of $rank_{ui}$. In the implicit case, the real rank of a positive sample is assumed to be 1, and the real rank of a negative item is assumed to be m. With these, we can differentiate the error function as follows:

$$\frac{\partial\,\mathrm{MSRE}}{\partial rank_i^{+}} = \frac{2}{m^2}\left(\frac{rank_i^{+}-1}{m}\right), \tag{3.30}$$

$$\frac{\partial\,\mathrm{MSRE}}{\partial rank_i^{-}} = \frac{2}{m^2}\left(\frac{rank_i^{-}}{m}-1\right). \tag{3.31}$$

Combining this with the other parts of the derivative chain in Equation 3.3 we get

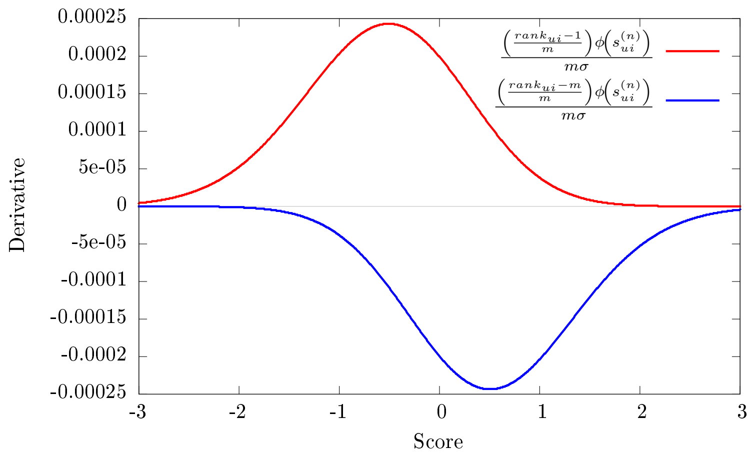


Figure 3.6: The coefficients of the update vectors used in SGD in the cases of positive and negatives samples with MSRE as objective function, assuming $m = 1000$, $\sigma = 1$ and $\mu = 0$.

$$\frac{\partial\,\mathrm{MSRE}}{\partial p_u} \propto -\sum_i \left(\frac{\left(\frac{rank_{ui}-rank_{ui}^{*}}{m}\right)\phi\left(s_{ui}^{(n)}\right)}{m\sigma} q_i\right), \tag{3.32}$$

where $rank_{ui}^{*} = 1$ if $i \in M^{+}$ and $rank_{ui}^{*} = m$ if $i \in M^{-}$.

3.4 Arbitrary gradient length function

There are multiple problems with the gradient functions described for ranking optimization in Section 3.3. The pointwise gradient function known to work well in practice is the gradient of the RMSE objective function (see Figure 3.8). This function works by pulling the scores to the desired value proportionately to their distance. This behavior grants the system an inherent stability, since the scores of the items are corrected if they grow too large. The fact that the gradient decreases around adds to this stability by making it harder for an update to “overshoot” the desired value.

If we compare this behavior to the behavior of the derivative of NDCG (see Figure 3.4), we can see that both of these stabilizing properties are missing. The size of the gradient vector is large on high scores, but small on low scores. This makes it so that it is very hard for a negatively rated item to rise to the top of the ranked list. The behavior of the top of the ranked list is chaotic, where even a single update can have a huge effect on the scores. The inherent correcting behavior of RMSE is also missing from the system - the size of the scores is not limited, they can grow arbitrarily large. The regularization term $\sum_i s_{ui}^2$ proposed by [14] (see 3.6) doesn’t really help with this, since the derivative of the regularization term is a first degree polynomial, meaning that all scores are pulled towards zero in steps linearly proportional to their size, which preserves their relative sizes. While both with AUROC and MSRE the gradient

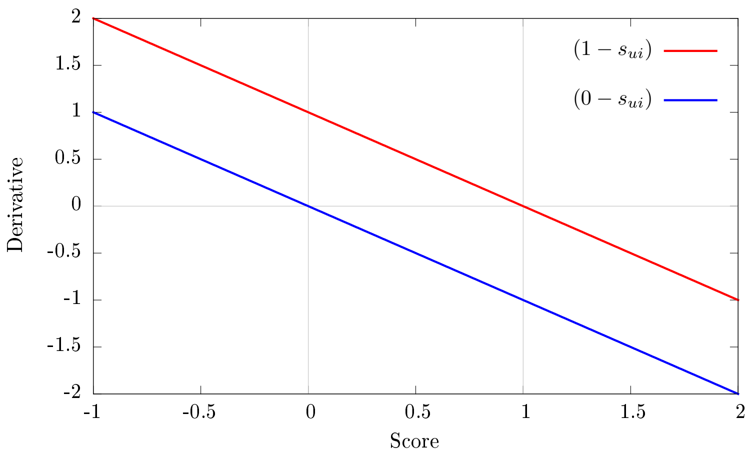


Figure 3.7: The coefficients of the update vectors used in SGD in the cases of positive and negatives samples with RMSE as objective function.

functions decrease near the desired values of the scores, the functions also decrease when moving towards the bottom of the ranking list, making it hard for low ranked items to raise.

An additional problem with the derived formulas is that for a fixed predicted score, the magnitude of the gradient changes throughout the learning process. First, in all of these formulas, the $\frac{1}{\sigma}$ coefficient tends to decrease if the system is not stabilized appropriately. Second, in online recommendation, m denotes the amount of items seen so far, which increases throughout the learning process. This affects all three of the derived formulas:

- In Equation 3.21 (derived from NDCG), the upper bound for $\frac{m}{(rank^{+}_{ui}+1)\log_2^2(rank^{+}_{ui}+1)}$ is $\mathcal{O}(m)$.
- In Equation 3.28 (derived from AUROC), the term $\frac{m}{m^{+}m^{-}}$ is $\mathcal{O}\left(\frac{1}{m}\right)$, assuming a constant rate of positive to negative samples.
- In Equation 3.32 (derived from MSRE), although the expression $\frac{rank_{ui}-rank^{*}_{ui}}{m}$ in the numerator equals either $\frac{m\left(1-\Phi\left(s^{(n)}_{ui}\right)\right)-m}{m}$ or $\frac{m\left(1-\Phi\left(s^{(n)}_{ui}\right)\right)-1}{m}$ (which is almost constant in m), the term $\frac{1}{m}$ present in the formula is obviously $\mathcal{O}\left(\frac{1}{m}\right)$.

We have to realize, that by using different objective function of the same general form of $\sum_i L(s_{ui})$, we are not changing the direction of the gradient in the multi dimension parameter space, only the length. In a general step of the SGD algorithm, by changing the objective function, we only change the step size coefficient $\frac{\partial L}{\partial s_{ui}}$:

$$(u, i) \leftarrow \text{random} \tag{3.33}$$

$$p_u \leftarrow p_u + \gamma\left(\left(\frac{\partial L}{\partial s_{ui}}\right) q_i - \lambda p_u\right) \tag{3.34}$$

$$q_i \leftarrow q_i + \gamma\left(\left(\frac{\partial L}{\partial s_{ui}}\right) p_u - \lambda q_i\right). \tag{3.35}$$

For traditional gradient descent, multiple methods exists for choosing the update step size, such as fixed step size, *exact line search* or *backtracking line search*. In recommender systems' related literature, generally the length of the gradient of the objective function is used for SGD, multiplied by a constant learning rate. It is possible however to choose any suitable function for this purpose.

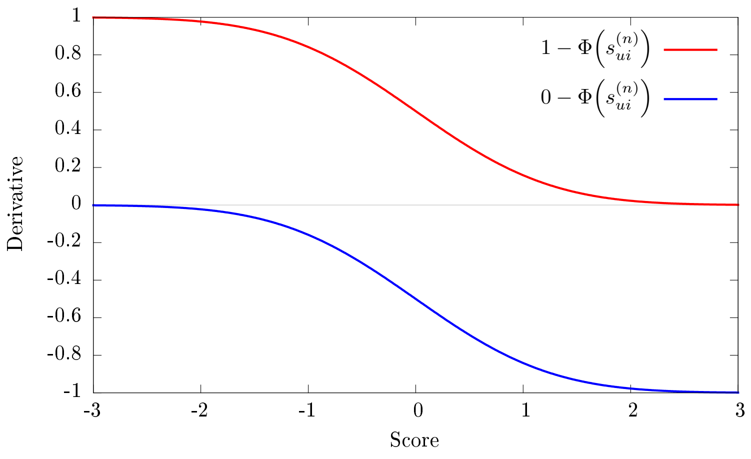


Figure 3.8: The coefficients of the update vectors used in SGD in with the relative rank error.

One function we examined is based on the Gaussian ranking idea presented in Section 3.1. This function we called *relative rank error* or *RRE* is defined as

$$\text{RRE} = \left| \frac{rank^*}{m} - \Phi\left(s_{ui}^{(n)}\right) \right| . \tag{3.36}$$

It has the advantageous property that the gradient nearly vanishes near the desired values. This function can be further tuned by applying a weight function to the error function.

Another concept we explored was to give different learning rates for negative and positive samples. These coefficients occur naturally in Equations 3.18 and 3.25, since we can choose any arbitrary convex combination of two expressions. This idea is also closely related to the concept of confidence described in Section 2.2.2, where the algorithm introduced by Koren et al. considers negatives samples less reliable by giving them significantly less weight in the training process.

Chapter 4

Experiments

We have experimented in detail with some of the ideas presented in Section 3. In Section 4.1, we describe the experimental settings, our evaluation methodology and the used dataset. In Sections 4.2, 4.3 and 4.4 we present the results of classic methods, the results our experiments with NDCG based objective functions and the results our experiments using AMF.

4.1 The used dataset and testing methodology

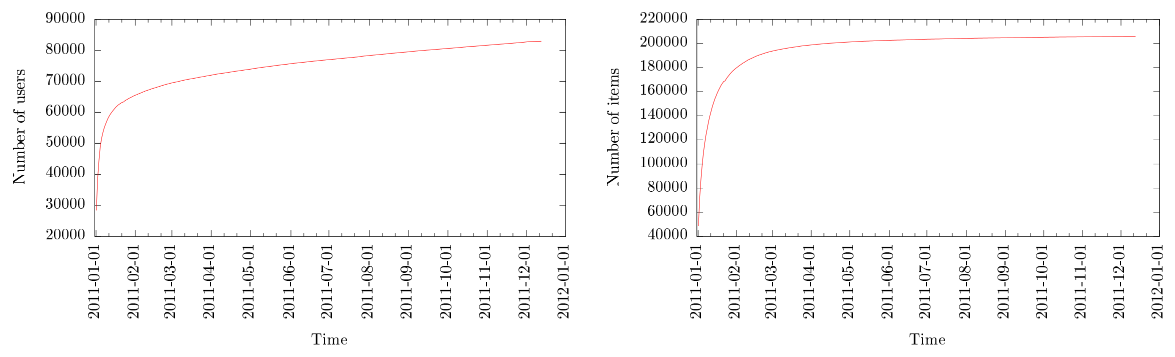


Figure 4.1: The growth of the number of seen users and items with time.

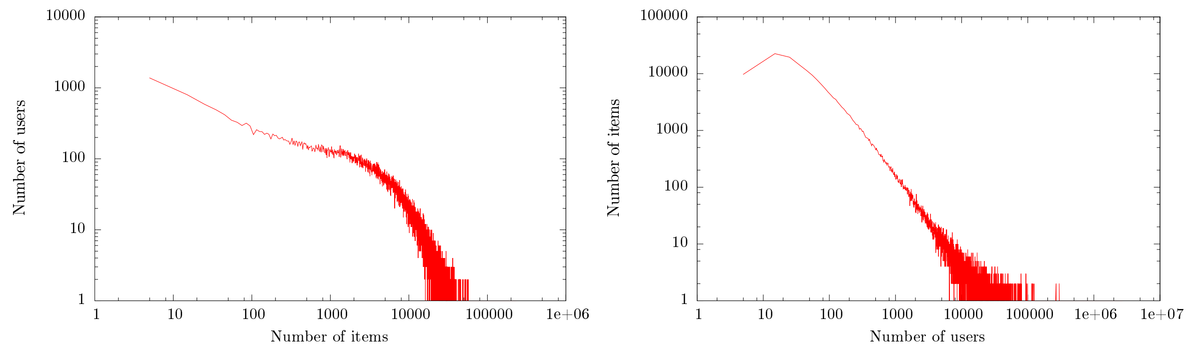


Figure 4.2: The distribution of the number of user-item relations for users and items.

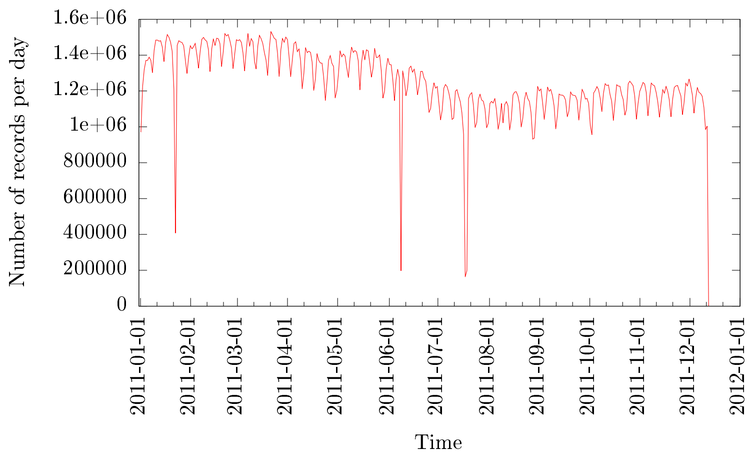


Figure 4.3: The number of daily records over time.

For our experiments, we used a 16GB dataset from LastFM, an internet based music streaming service, containing 346 days worth of data from the year 2011. The dataset consists of about 430 million implicit records, and features around $200,000$ artists and $82,000$ users. A record indicates the time a user has first listened to an artist. Figure 4.1 shows the number of users in the system over time. Figure 4.2 shows the distribution of the number of user-item relations for users and for items. Figure 4.3 shows the number of records per day in the dataset.

For testing, we used a C++ recommender systems framework developed in SZTAKI. In our experiments, we processed the data in an online fashion. With each new training record, we first calculated the prediction of the system for that particular user-item pair, recorded the rank of the item on the toplist of the not-yet rated items of the user, and then used the record for training the model. After processing a record, we employed negative sample generation, choosing negative samples from the pool of item indexes not-yet rated by the user.

The results were evaluated by processing the recorded item-ranks and summing the implicit NDCG formula

$$\text{NDCG @100} = \begin{cases} \frac{1}{\log_2(rank_{ui}+1)} & \text{if } rank_{ui} \leq 100, \\ 0 & \text{otherwise,} \end{cases} \tag{4.1}$$

and then averaged over suitable time intervals. The parameter tests were conducted by running the algorithms with different parameter values, and plotting the cumulative NDCG average against the tested parameter values.

The models were initialized generally with random numbers from the distribution $U([-0.01, 0.01])$. The factor dimension was chosen to be 10.

4.2 Baselines

In our tests, we have compared the results of experimental methods to those of classic methods. For this purpose, we run SGD with RMSE as objective function on the classic matrix factorization model and on the AMF model. As we can see on figure 4.4, AMF outperforms classic MF initially, but performs slightly worse over the one year interval. This occurs because by design, AMF can incorporate new users into the system much faster than classic MF. As seen on Figure 4.1, the number of users grows rapidly in the first month of the interval, and only slightly over the remaining 11 months.

Figures 4.5 and 4.6 show the performance of the baseline methods with different parameter values. Regularization was not used, because the relatively low number of factor dimensions makes it unnecessary, as it prevents overfitting.

4.3 Experiments using NDCG

We have implemented the function seen in Equation 3.21. For estimating the distribution mean, we stored and incrementally updated the value $\sum_{i \in M} q_i$. With this, we can calculate the exact value of μ_u, the mean of the scores of a given row of the matrix as

$$\mu_u = \frac{p_u \left(\sum_{i \in M} q_i\right)^T}{m}. \tag{4.2}$$

Unfortunately, this is not possible in the case of variance. To estimate the distribution variance we used the sample variance. We have tried two sampling methods. One was to store the last l positive items from the training data in a queue, and use them for sampling scores from the row corresponding to a particular user. The other was to sample l items randomly from the users present in the system. This was done efficiently by storing all item indices in a vector and employing a variant of the Fisher-Yates shuffle algorithm:

```
$\forall i \in \{0, 1, \ldots, (l-1)\}$ :
    $j \leftarrow$ uniform random integer from the interval $[i, m-1]$
    swap$(a[i], a[j])$,
```

where a is the array of users, and swap is a method for exchanging two elements of an array. After the shuffle, the first l elements of the array contain l uniformly chosen non-repeating samples. We have not seen noticeable differences in performance when comparing these two methods. Since the value σ is present in the denominator of the gradient function, the value of the formula is very sensitive to small σ values. To lessen the effect of sampling error, we decided to introduce a minimum (cutoff) value for σ, which was chosen to be 0.01.

In [14], Steck et al. suggest using a piecewise quadratic function for approximating the cumulative distribution function of the normal distribution:

$$quad(x) = \begin{cases} 0 & \text{if } x \leq -1 \\ (1+x)^2/2 & \text{if } -1 < x \leq 0 \\ 1-(1-x)^2/2 & \text{if } 0 < x < 1 \\ 1 & \text{if } 1 \leq x. \end{cases} \tag{4.3}$$

with scaling $\Phi(x) = quad(x/\sqrt{6})$. However, since with this approximation the expression $\frac{\phi(x)}{\Phi(x)}$ present in the gradient formula is highly inaccurate (see Figure 4.7), we instead used the Boost C++ library's implementation of ϕ and Φ.

Our initial results with NDCG as objective function were promising. As we can see on Figure 4.8, it achieved approximately half of the NDCG score of the baseline RMSE MF. We hoped that with proper parameter tuning, we could replicate the good performance of [14] in an online setting. However, this was proven to be not possible. As Figure 4.9 shows, different parameter values were not able to significantly improve the performance of the method. The results show, that it works best with small negative rates. This prompted us to experiment with different weight values for the negative samples. We ran two experiments: changing the negative rate with a lower negative sample coefficient, and changing the negative sample coefficient with a fixed negative rate (see Figure 4.10). We have also tried turning negative sample generation off altogether, and using the regularization coefficient from Equation 3.6 instead, which yielded quite poor results (see Figure 4.11).

The best results we were able to achieve by tuning these parameters was to outperform the baselines in the first few days (Figure 4.12). Unfortunately, this did not help the fact that the quality of the recommendations drastically decreases over the one year interval. We suspected that the poor long term performance of the algorithm is related to the fact described in Section 3.4 that particular coefficients in the formula change in magnitude throughout the learning process. In the case of NDCG, the most problematic is probably the $\frac{1}{\sigma}$ coefficient. Figure 4.13 shows the change of variance, average of the absolute values of the gradients and average of the absolute values of the predicted scores for the training samples.

With the growth of σ, the length of the gradient diminishes over time. This is the reason for the behavior of the system in regards to cumulative average performance when increasing the learning rate, see Figure 4.9. To try and correct this behavior, we experimented with periodically scaling back every row of P and Q so that the average of their norms would be equal to a fixed value. We experimented with different periods and different target values, with no significant improvements in prediction accuracy. Figure 4.14 shows the change of variance, average of the absolute values of the gradients and average of the absolute values of the predicted scores for the training samples when the latent vectors are scaled back every 10000 update to an average norm of 0.1. Figure 4.15 shows that this technique was able to achieve a non-significant improvement.

Ultimately, we concluded that this objective function is not able to perform comparably to the classic RMSE objective function. This is probably due to the shape of the gradient length function seen in Figure 3.4, which prevents the model from learning from the training data efficiently.

4.4 Experiments using AMF

Discouraged by the ineffectiveness of the NDCG based objective function, we started experimenting with one based on mean square rank error. This time, instead of sampling the scores to approximate the variance, we decided to assume a fixed value for both the average and the variance, and to attempt to enforce this value. The way of enforcing was by including the regularization term $\sum_{i \in M} s^2$ from Equation 3.6. However, for better performance, we modified it so that it only sums over the negative items of a user: $\sum_{i \in M^-} s^2$, thus it only affects the gradients of the negative samples. We called this kind of regularization *score regularization*, and the coefficient α in Equation 3.6 *score regularization coefficient*. Also, for reasons described in Section 3.4, we had to ignore the $\frac{1}{m}$ coefficient in the gradient formula, since this would have made impossible for the model to perform well throughout the given time interval.

This approach turned out to perform quite well, when applied with low negative sample coefficient and high negative rate. As seen on Figure 4.16, it managed to beat both baselines at the start of the year, and had good performance all the way.

In these tests, we assumed a contant variance of 0.01 and an expected value of 0. Figure 4.18 shows the distribution of scores after processing 20 and 60 days worth of training data. The sample mean and variance of these distributions are, respectively, $\mu_{20} = -0.000243, \sigma_{20} = 0.000003$ and $\mu_{60} = -0.000312, \sigma_{60} = 0.000003$. The figure and the sample variance values indicate that the training algorithm did not succeed in enforcing the desired distribution. This makes it unlikely that the good performance was due to the derived objective function.

Figure 4.17 shows the performance of the model when changing different parameter values. With more extensive parameter tuning, the results seen on Figure 4.16 could even be improved upon slightly. However, it is suspect that the good performance is not due to the effectiveness of the objective function, rather the low negative coefficient and high negative rate. To come to a conclusion about the viability of the model, we'll need further testing in this regard.

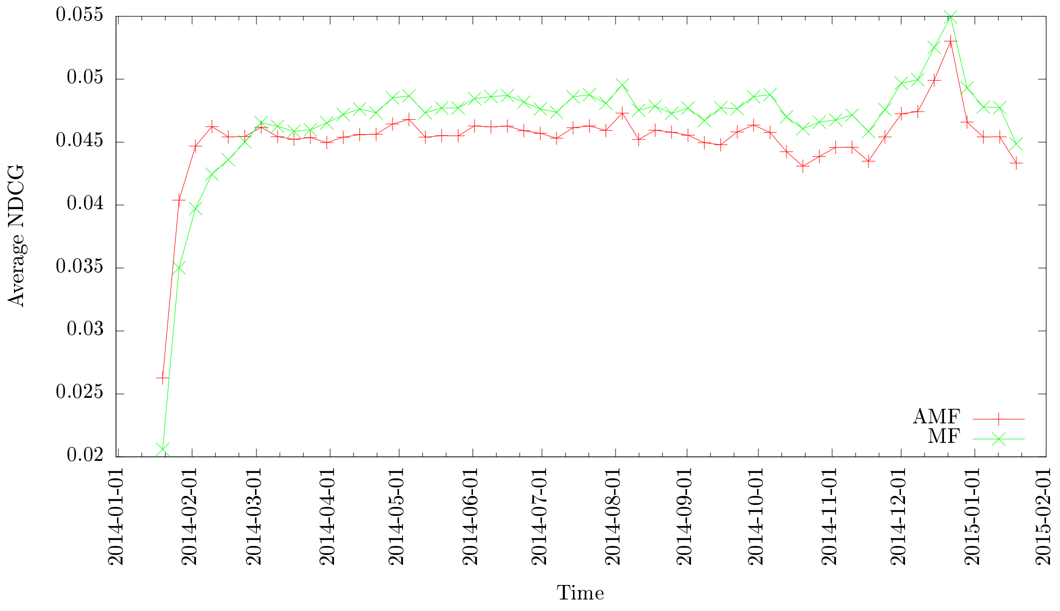


Figure 4.4: Performance of SGD on the classic matrix factorization model and the AMF model. The NDCG values are averaged over one week intervals. Parameters of MF: learning rate= 0.14, negative rate= 99. Parameters of AMF: learning rate= 0.11, negative rate= 110.

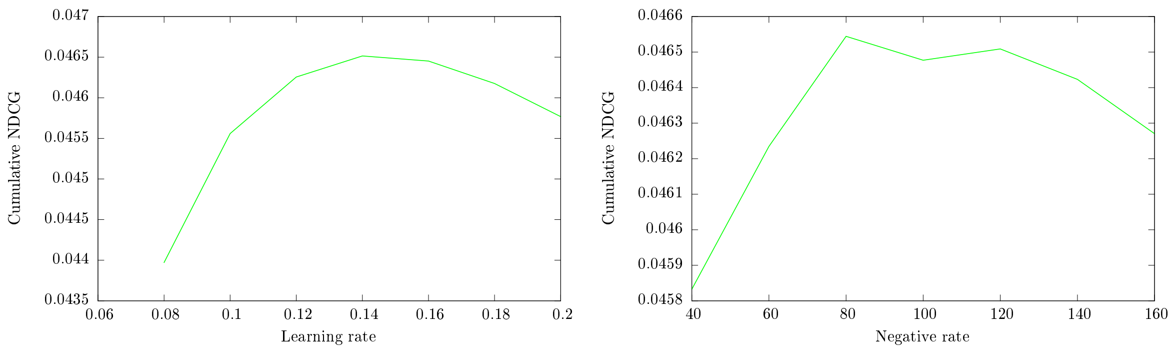


Figure 4.5: Performance of SGD on the classic MF model over a one year interval with different parameter values.

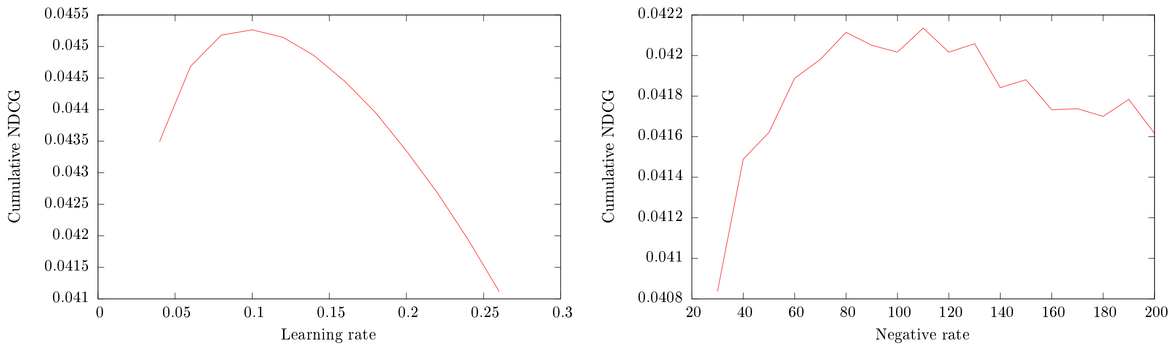


Figure 4.6: Performance of SGD on the AMF model over a one year interval with different parameter values.

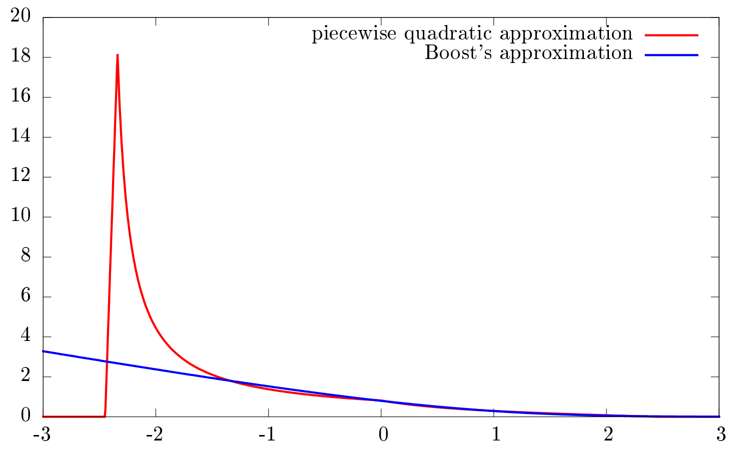


Figure 4.7: Different approximations for $\frac{\phi(x)}{\Phi(x)}$.

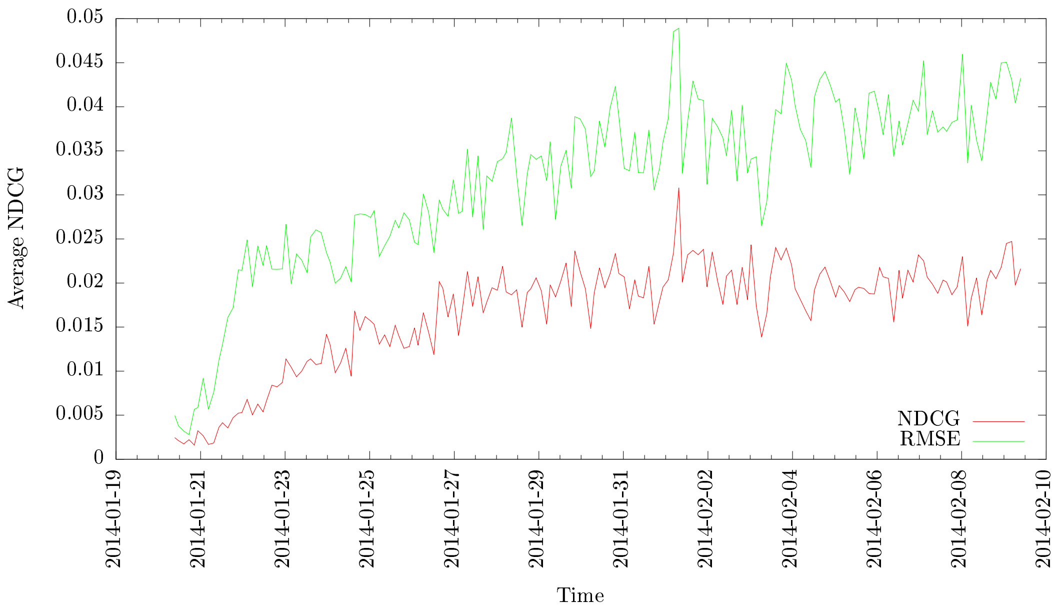


Figure 4.8: The first result of optimizing for NDCG, over a 20 day interval.

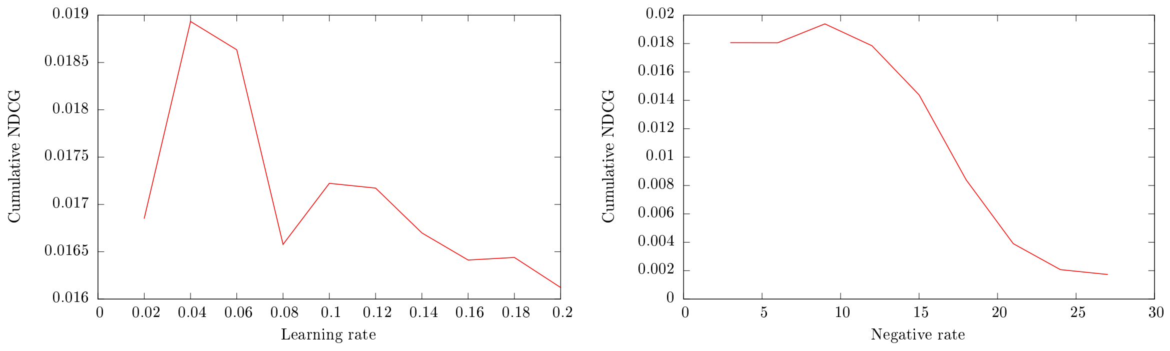
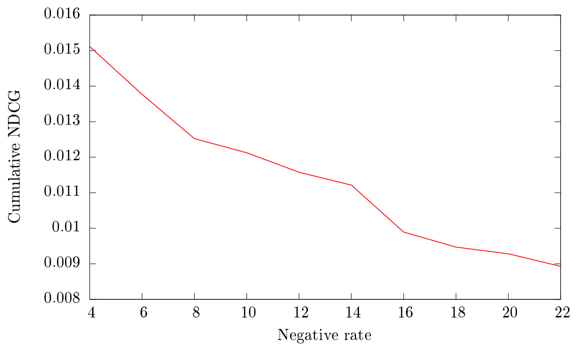


Figure 4.9: Performance of SGD with NDCG as objective function over a one year interval with different parameter values.



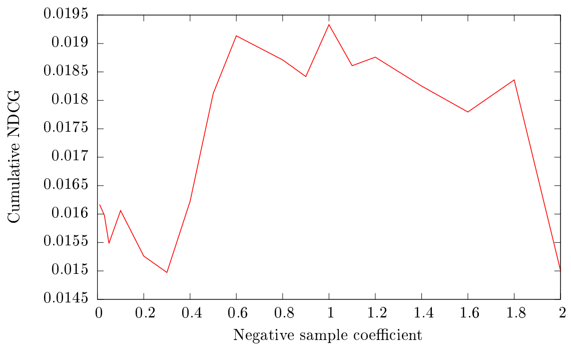


Figure 4.10: Changing the negative rate with a lower negative sample coefficient (0.1), and changing the negative sample coefficient with a fixed negative rate (10).

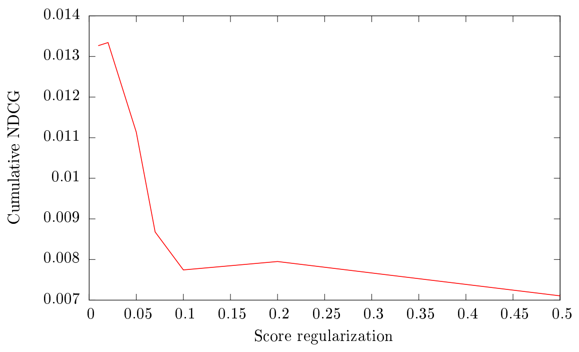
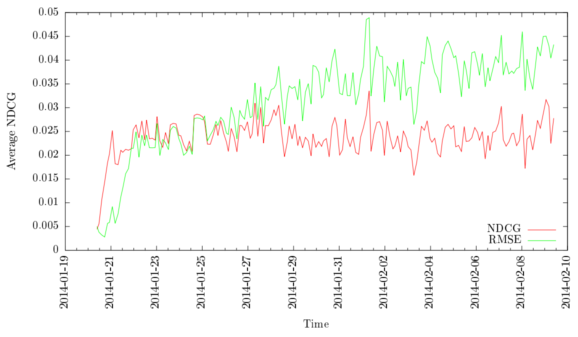


Figure 4.11: Performance of SGD with NDCG as objective function, with different parameter values for score regularization.



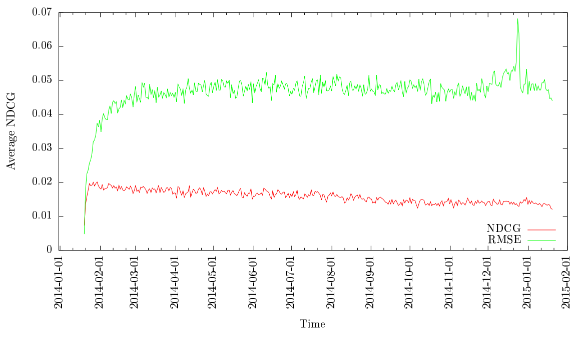


Figure 4.12: Outperforming the baseline in the short term did not help with the poor long term performance of the method.

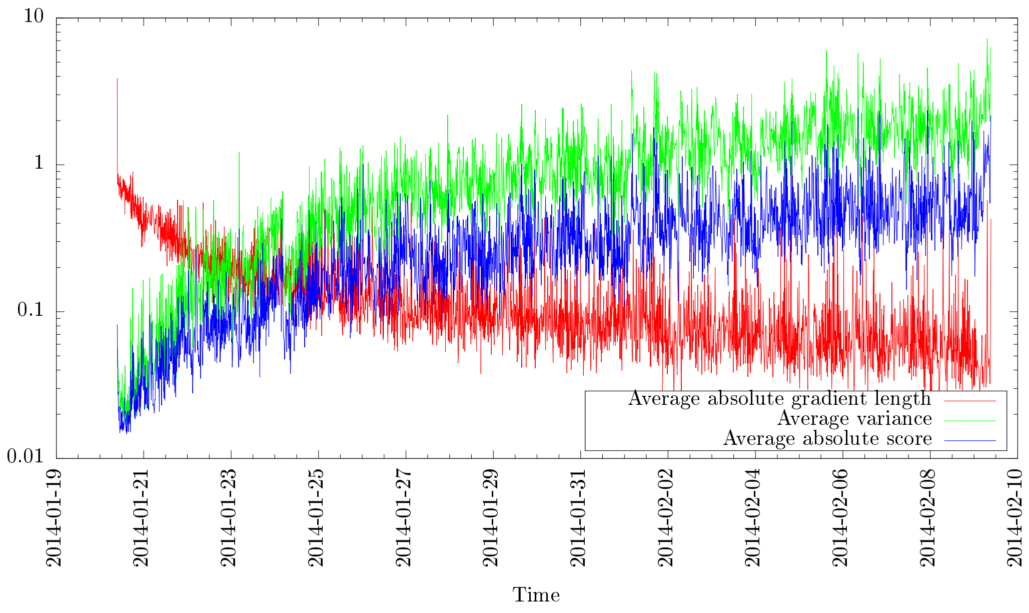


Figure 4.13: The change of variance, average of the absolute values of the gradients and average of the absolute values of the predicted scores for the training samples with NDCG as objective function, over a 20 days interval.

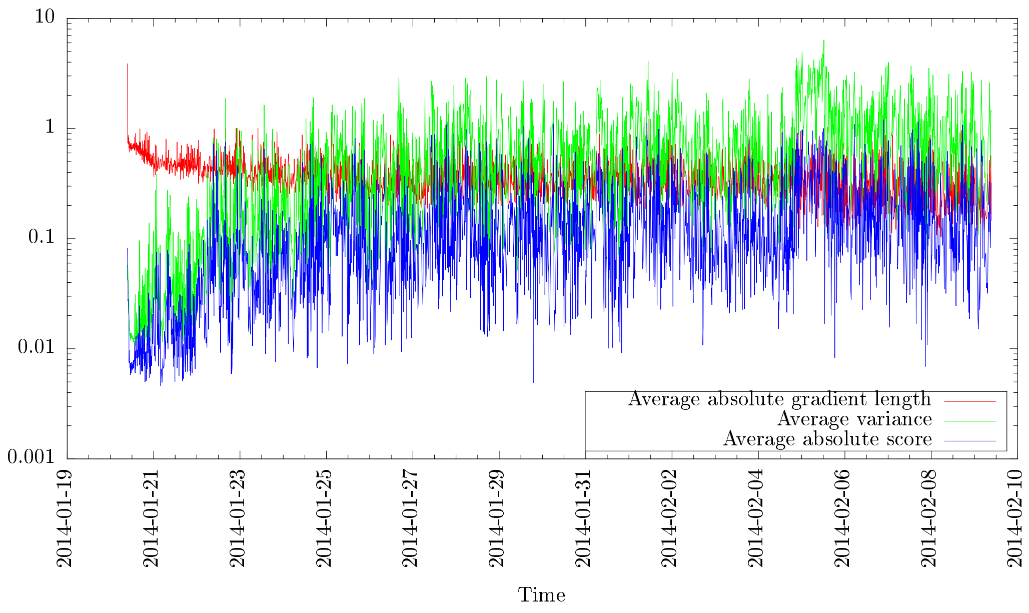


Figure 4.14: The change of variance, average of the absolute values of the gradients and average of the absolute values of the predicted scores for the training samples with NDCG as objective function, over a 20 days interval, when scaled back every 10000 updates to an average norm of 0.1.

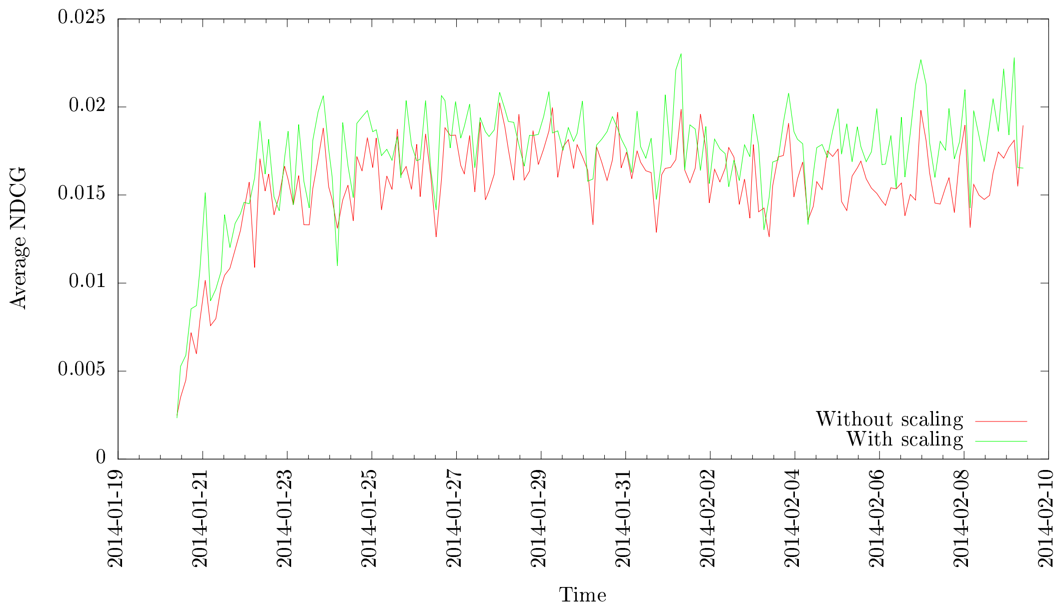


Figure 4.15: Prediction accuracy with and without periodic scaling of the latent vectors when optimizing for NDCG.

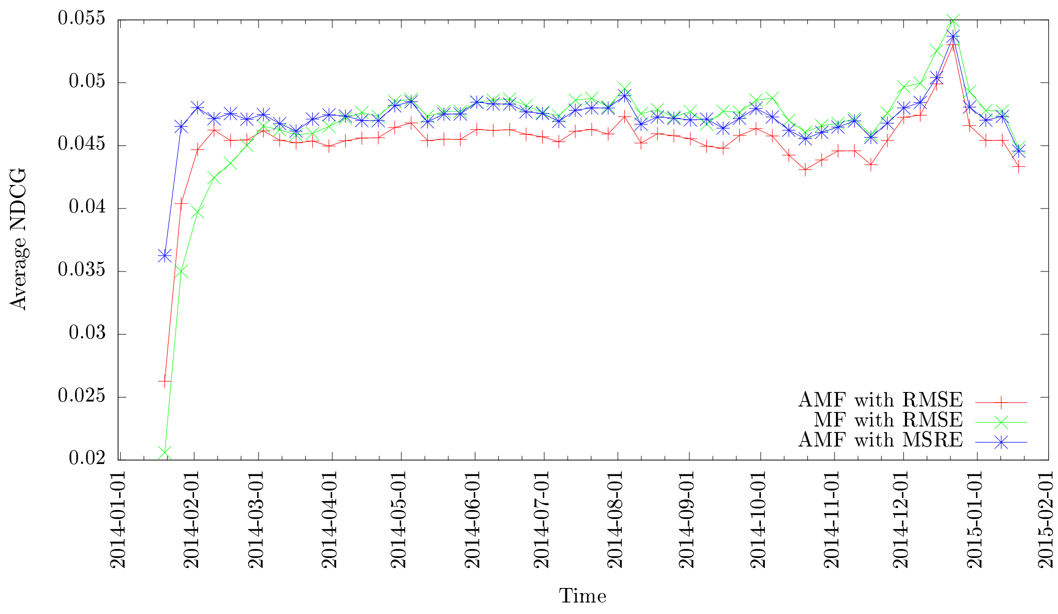


Figure 4.16: Comparing AMF with MSRE as objective function to the baselines. Parameters of the model: learning rate: 0.2, negative rate: 500, positive sample coefficient:7, negative sample coefficient: 0.01, score regularization coefficient: 1.

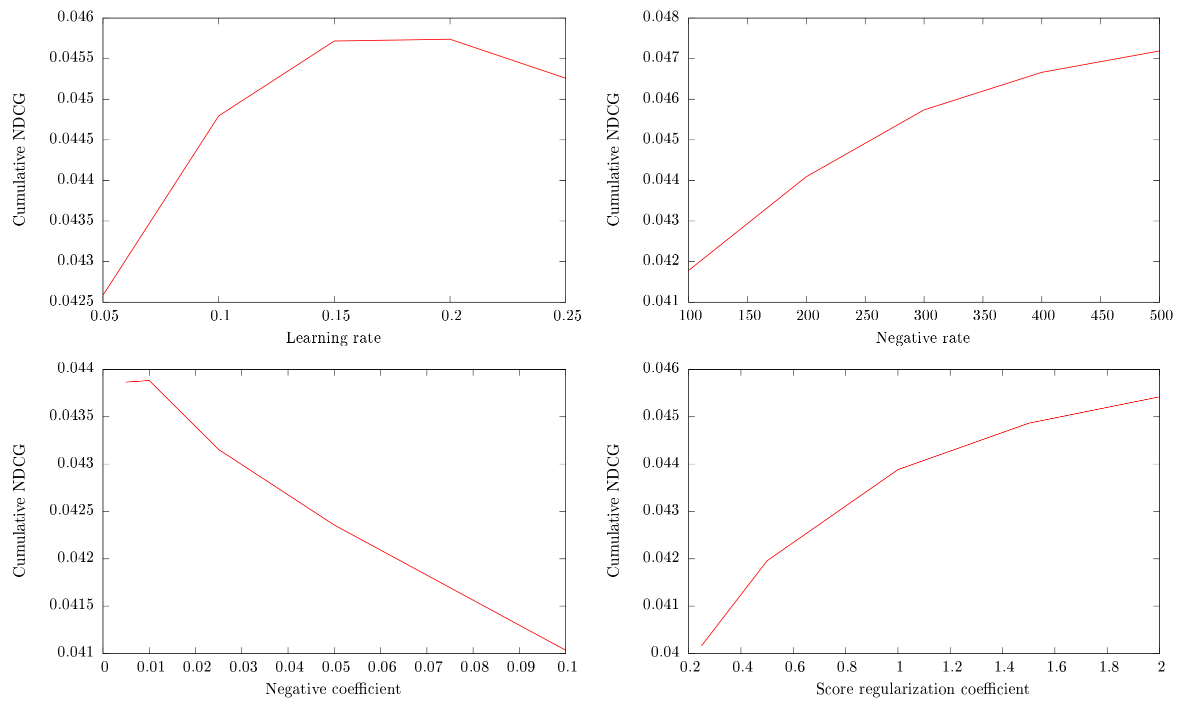


Figure 4.17: Cumulative average NDCG performance of AMF with MSRE as objective function, when changing the model parameters.

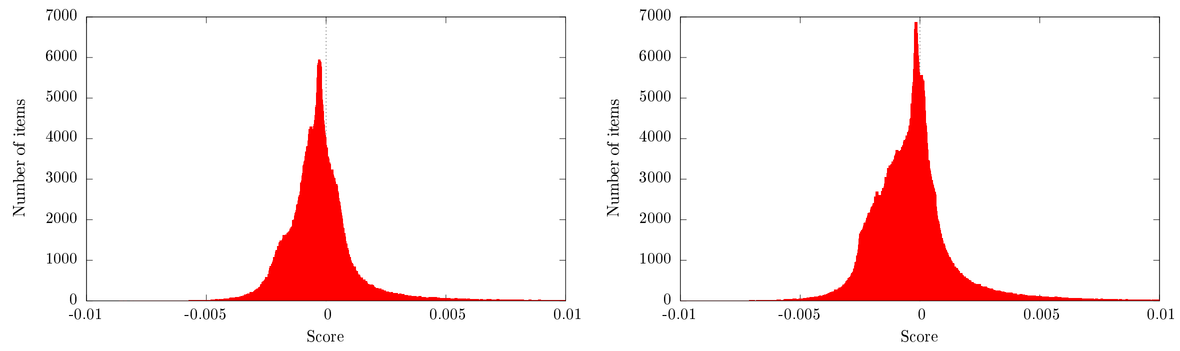


Figure 4.18: Distribution of scores predicted by AMF with RMSRE as objective function, after 20 days of training (left) and after 60 days of training (right).

Summary

In this thesis, our goal was to summarize general concepts, models and algorithms related to recommender systems, and to describe our research into optimizing for ranking prediction evaluation functions. In Section 1, we described basic terminology and concepts. In Section 2, we described the matrix factorization model in detail, including objective functions, popular algorithms and techniques. In Section 3, we summarized the results presented in [14], and our own approach to applying these results for online recommendation. This included deriving exact gradient formulas for the SGD algorithm based on the results of Steck et al., their analysis, and also presenting novel gradient functions based on Gaussian ranking. Finally, in Section 4, we presented our own experimental results, including detailed analysis of the ineffectiveness of NDCG based objective functions and the results of other, more effective methods.

We concluded, that – as derived from definitions – Gaussian ranking based objective functions are not viable for online recommendation. While with slight modifications we were able to achieve comparable results to the baselines, futher research is needed to determine the exact reasons for this good performance.

Bibliography

[1] Steffen Rendle et al. „BPR: Bayesian Personalized Ranking from Implicit Feedback". In: *CoRR* abs/1205.2618 (2012). URL: http://arxiv.org/abs/1205.2618.

[2] Y. Koren, R. Bell and C. Volinsky. „Matrix Factorization Techniques for Recommender Systems". In: *Computer* 42.8 (Aug. 2009), pp. 30–37. ISSN: 0018-9162. DOI: 10.1109/MC.2009.263.

[3] R. M. Bell and Y. Koren. „Scalable Collaborative Filtering with Jointly Derived Neighborhood Interpolation Weights". In: *Seventh IEEE International Conference on Data Mining (ICDM 2007).* Oct. 2007, pp. 43–52. DOI: 10.1109/ICDM.2007.90.

[4] Christian Desrosiers and George Karypis. "A comprehensive survey of neighborhood-based recommendation methods". In: *Recommender systems handbook.* Springer, 2011, pp. 107–144.

[5] Robert M. Bell and Yehuda Koren. „Lessons from the Netflix Prize Challenge". In: *SIGKDD Explor. Newsl.* 9.2 (Dec. 2007), pp. 75–79. ISSN: 1931-0145. DOI: 10.1145/1345448.1345465. URL: http://doi.acm.org/10.1145/1345448.1345465.

[6] Alan Herschtal and Bhavani Raskutti. „Optimising area under the ROC curve using gradient descent". In: *Proceedings of the twenty-first international conference on Machine learning.* ACM. 2004, p. 49.

[7] Yue Shi, Martha A. Larson and Alan Hanjalic. „List-wise Learning to Rank with Matrix Factorization for Collaborative Filtering". In: *RecSys '10: Proceedings of the fourth ACM conference on Recommender systems.* ACM. Barcelona, Spain: ACM, 2010, pp. 269–272. ISBN: 978-1-60558-906-0.

[8] Carl Eckart and Gale Young. „The approximation of one matrix by another of lower rank". In: *Psychometrika* 1.3 (1936), pp. 211–218. URL: http://EconPapers.repec.org/RePEc:spr:psycho:v:1:y:1936:i:3:p:211-218.

[9] Badrul Sarwar et al. *Application of dimensionality reduction in recommender system-a case study.* Tech. rep. DTIC Document, 2000.

[10] Badrul Sarwar et al. „Incremental singular value decomposition algorithms for highly scalable recommender systems". In: Citeseer.

[11] István Pilászy, Dávid Zibriczky and Domonkos Tikk. „Fast als-based matrix factorization for explicit and implicit feedback datasets". In: *Proceedings of the fourth ACM conference on Recommender systems.* ACM. 2010, pp. 71–78.

[12] Y. Hu, Y. Koren and C. Volinsky. „Collaborative Filtering for Implicit Feedback Datasets". In: *Data Mining, 2008. ICDM '08. Eighth IEEE International Conference on.* Dec. 2008, pp. 263–272. DOI: 10.1109/ICDM.2008.22.

[13] Arkadiusz Paterek. „Improving regularized singular value decomposition for collaborative filtering". In: *Proc. KDD Cup Workshop at SIGKDD'07, 13th ACM Int. Conf. on Knowledge Discovery and Data Mining.* San Jose, CA, USA, 2007, pp. 39–42. URL: http://serv1.ist.psu.edu:8080/viewdoc/summary;jsessionid=CBC0A80E61E800DE518520F9469B2FD1?doi=10.1.1.96.7652.

[14] Harald Steck. „Gaussian Ranking by Matrix Factorization". In: *Proceedings of the 9th ACM Conference on Recommender Systems.* RecSys '15. Vienna, Austria: ACM, 2015, pp. 115–122. ISBN: 978-1-4503-3692-5. DOI: 10.1145/2792838.2800185. URL: http://doi.acm.org/10.1145/2792838.2800185.

[15] Yehuda Koren. „Collaborative Filtering with Temporal Dynamics". In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* KDD '09. Paris, France: ACM, 2009, pp. 447–456. ISBN: 978-1-60558-495-9. DOI: 10.1145/1557019.1557072. URL: http://doi.acm.org/10.1145/1557019.1557072.

[16] Harald Steck. „Evaluation of Recommendations: Rating-prediction and Ranking". In: *Proceedings of the 7th ACM Conference on Recommender Systems.* RecSys '13. Hong Kong, China: ACM, 2013, pp. 213–220. ISBN: 978-1-4503-2409-0. DOI: 10.1145/2507157.2507160. URL: http://doi.acm.org/10.1145/2507157.2507160.

[17] Asela Gunawardana and Guy Shani. „A Survey of Accuracy Evaluation Metrics of Recommendation Tasks". In: *J. Mach. Learn. Res.* 10 (Dec. 2009), pp. 2935–2962. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1577069.1755883.