TEL AVIV UNIVERSITY
אוניברסיטת תל-אביב
THE IBY AND ALADAR FLEISCHMAN
FACULTY OF ENGINEERING
הפקולטה להנדסה
על שם איבי ואלדר פליישמן

Advanced Techniques for Latent Factor Recommender Systems

A thesis submitted towards the Degree of

Doctor of Philosophy

by

Noam Koenigstein

Under the supervision of Prof. Yuval Shavitt

Tel Aviv University

August 2013

Table of Contents

Acknowledgments **iv**

List of Figures **1**

List of Tables **2**

1: Introduction **5**

1.1 Thesis Organization 6

1.2 Notation 7

2: Recommenders Based on Matrix Factorization **8**

2.1 A Basic Model 8

2.2 Learning Parameters 9

2.2.1 Minimizing Squared Loss 10

2.2.2 A Bayesian Formulation 12

3: Modeling with Temporal Dynamics and Item Taxonomy **16**

3.1 The KDD-Cup'11 Data Challenge 16

3.2 The Yahoo! Music Dataset 17

3.2.1 Characterizing the Yahoo! Music Dataset 17

3.2.2 Items Taxonomy 20

3.3 Modeling Bias Patters 21

3.3.1 The Need for Biases 21

3.3.2 A Basic Bias Model 22

3.3.3 Taxonomy biases 23

3.3.4 User Activity Sessions 24

3.3.5 Item Temporal Biases 26

3.3.6 A Full Bias Model . . . 27
3.4 Modeling Personalization . . . 28
3.4.1 Utilizing Taxonomy . . . 28
3.4.2 Personalizing Listening Sessions . . . 29
3.4.3 Stochastic Gradient Descent (SGD) Optimization . . . 30
3.4.4 Tuning Meta-parameters with The Nelder-Mead Algorithm . . . 31
3.5 Evaluation . . . 32
3.5.1 Visualizing the Latent Space . . . 34

4: Efficient Retrieval of Recommendations 37

4.1 Problem Definition . . . 38
4.2 Nearest Neighbor Search Algorithms . . . 40
4.2.1 Nearest-neighbor Search in a Metric Space . . . 40
4.2.2 Cosine similarity . . . 41
4.2.3 Locality Sensitive Hashing (LSH) . . . 41
4.2.4 Maximum Dot-products . . . 42
4.3 Fast Exact Retrieval Using Metric Trees . . . 42
4.3.1 Metric Trees . . . 42
4.3.2 Branch-and-bound Algorithm . . . 43
4.4 Fast Approximate Retrieval by Clustering Users . . . 48
4.4.1 Approximation Error Bound . . . 50
4.5 Experiments and Evaluations . . . 52
4.5.1 Datasets . . . 52
4.5.2 Exact RoR . . . 54
4.5.3 Approximate RoR . . . 55
4.5.4 Existing Best-Match Algorithms . . . 59

5: Item Based Recommendations 61

5.1 Related Work . . . 62
5.2 Modeling Pairwise Item Relations . . . 63

5.2.1 Likelihood Maximization 64
5.3 Fast Retrieval 65
5.4 Empirical Study 66
5.4.1 Dataset Construction 66
5.4.2 Baselines 67
5.4.3 Accuracy Results 68
5.4.4 Fast Retrieval Results 69
5.5 Future Work 70

6: Additional Work **71**

6.1 Additional Research on Recommender Systems 71
6.1.1 Real World Recommender Systems 71
6.1.2 Feature Selection for Recommender Systems 71
6.1.3 Group Recommendations 72
6.2 Music Information Retrieval in Peer-to-Peer Networks 73
6.2.1 Trend Prediction Based on P2P Queries 73
6.2.2 Collaborative Filtering Based on P2P Networks 74

References **75**

Acknowledgments

I would like to thank all those who helped, supported and advised me throughout or at different stages of my degree. First and foremost, I thank my advisor, Yuval Shavitt, for the guidance and mentoring through both my masters and Doctoral degrees. You always had an enlightening advise and insights on algorithmic or academic questions, as well as any other issue at question.

Thanks to Yehuda Koren, who introduced me to recommendation systems and guided me during my six moths internship at Yahoo! Research and ever since after. It has been a breakthrough opportunity for me and I attribute much of my later professional success to you.

While at Yahoo! Labs, I was also fortunate to be guided by Gideon Dror. I really enjoed working with you and gained much from your experience and inputs.

Special thanks to my Austrian friends Markus Schedl and Peter Knees for all the various collaborations in ISMIR and AdMIRe.

I want to thank Nir Nice, my manager at Microsoft for much support in balancing my PhD. with full time employment at Microsoft. With your support I was able to both peruse and complete my degree while gaining real world experience in developing recommendation algorithms for Xbox.

To my colleague at Microsoft Ulrich Paquet – for the last two years in which we developed Xbox recommendation algorithm serving more than 50 million users world wide. I learned much from you, and I appreciate our friendship.

Parikshit Ram for your dedication through our long distance research collaboration.

I would like to thank everyone at the DIMES lab in Tel Aviv University for the collaborations, technical support and most importantly friendship: Udi and Ela Weinsberg, Noa Zilberman, Ido Blutman and Boaz Harel.

I also want to thank other collaborators: Pavel Gurvich, Gert Lanckriet, Brian McFee,

Tim Pohle, Nir Schleyen and late Tomer Tankel.

Finally, I like to thank my parents Michael and Tehila and the rest of the family: David, Gerta, Dvir, Rimon, Geffen and Gordon.

List of Figures

2.1 Stochastic Gradient Descent Optimization 11

2.2 Alternating Least Squares Optimization 13

2.3 A Graphical Model Representation 14

3.1 The Yahoo! Music Dataset Statistics 18

3.2 Session Length Effect on User Biases 25

3.3 Items Temporal Basis Functions 27

3.4 Breaking Down RMSE 34

3.5 2-Dimensional Latent Space of Yahoo! Music 36

4.1 Metric Trees 43

4.2 Metric-tree Construction 44

4.3 Bounding Inner-product With a Ball 45

4.4 Metric-tree Search 47

4.5 Bounding the Approximation Error of User-cones 50

4.6 Approximate Retrieval with User-Cones 51

4.8 $MedianRank(K)$ of The Adaptive Algorithm 57

List of Tables

3.1 RMSE of Different Models 33
3.2 RMSE of Different Item Types 34

4.1 Speedup of The Exact Retrieval Algorithm 55
4.2 Speedup of The Approximate Retrieval Algorithm 56
4.3 Precision at K vs. l_2 Distance 59
4.4 Precision at K Based on Best Cosine Similarity Matching 59

5.1 EIR Evaluation - Datasets Statistics 67
5.2 Mean Percentile Rank of EIR 69
5.3 Speedup Values for EIR with 50 Dimensions 69

Abstract

Recommender systems are increasingly used in industry for predicting taste and preferences. Algorithms based on Matrix Factorization (MF) are one of the most efficient and accurate approaches to collaborative filtering. In these models users and items are embedded into a lower dimensional latent space in which the affinity between a user and an item is determined by the inner product of their representative vectors.

This thesis spans across three main tiers of MF models. First, we examine the basic setting and compare different approaches for defining and learning the problem. We then present some novel enhancements for modeling temporal dynamics and taxonomies within the MF framework. We focus on a model designed for the Yahoo! Music dataset that was developed as part of the preparations for KDD-Cup'11 on music recommendations.

The second tier of this thesis deals with a scalability issue at the retrieval phase – after the training has concluded. We introduce a difficulty caused by the need to efficiently compute a large number of inner products between user and item vectors. We were surprised to discover that while this is a common problem to any large scale recommender based on Matrix Factorization, the problem of efficient retrieval in an inner product space has hardly been addressed in literature. We therefore suggest a novel data structure for efficient retrieval of the k-nearest neighbors where similarity is based on the inner-product.

The last tier of this thesis deals with item oriented recommendations, but also touches on the scalability issue from a different angle. We abandon the traditional approach of MF models which tries to model both users and items, and instead propose a model that learns probabilities for two items to be co-consumed together. This work is motivated by many real world scenarios in which users long term history does not exists or irrelevant to her current interests. The typical sparseness of collaborative filtering datasets prevents establishing reliable similarity relations for many item pairs in the long tail. We thus

propose a global optimization procedure that utilizes both direct and indirect relations between items in order to encode co-consumption probabilities in a latent Euclidean space. The use of a Euclidean space rather than an inner product space bypasses the scalability issue from above by facilitating the use of many well studied nearest neighbor retrieval algorithms.

I Introduction

Collaborative Filtering (CF) recommender systems based on latent factor models have repeatedly demonstrated superior results when compared to other algorithms (e.g., neighborhood models and restricted Boltzmann machines) [1, 2]. This thesis focuses on latent factor models for recommendation systems, sometimes known also as Matrix Factorization (MF) models. The main contribution of this thesis is threefold: First, we present state of the art extensions to the basic MF model that can capture more signals such as taxonomy and different temporal dynamics. Specifically, we are first to show a model that can utilize taxonomy to improve predictions accuracy. This work falls inline with state of the art research in the field, in which different variants of MF models are proposed in order to improve aspects such as accuracy, recommendation quality, or scalability.

Second, we introduce a scalability challenge common to any MF model with regard to the retrieval of the recommendations. Retrieval of recommendation requires computing a very large number of inner products between user vectors and item vectors which strains CPU resources and cause high latency in online systems. While this task is essential in any large scale real world recommender, the problem was hardly discussed in the academic literature. The core of this study deals with finding top-k vectors that maximize the inner product with some other "query" vector. We are first to introduce algorithmic solutions (both exact and approximate) to this problem.

Third, we discuss item-oriented recommendations in which the recommendations are computed based solely on the small set of items the user is *currently* considering. Unlike the "traditional" case in which the algorithm models both users and items, our approach models only items based on a history of short term interactions. Furthermore, our item-based model encodes items similarity via Euclidean proximity (rather than inner product

similarity) and therefore overcomes the aforementioned retrieval challenge by enabling the use of the plethora of Euclidean nearest neighbor retrieval algorithms (e.g., metric trees, local sensitive hashing, etc.).

1.1 Thesis Organization

This thesis begins with a comprehensive overview of latent factor model discussed in Chapter 2. We formulate the challenge of a CF recommender system and present different learning algorithms to infer latent parameters.

In Chapter 3 we extend the basic model from Chapter 2 and present a sate of the art model that captures signals such as taxonomy and different temporal signals to improve accuracy. Different parts of this research has been published in [2, 3, 4]. The initial part of this work was done in preparation to the KDD'11-Cup competition organized by Yahoo! Research under the guidance of Gideon Dror and Yehuda Koren. The chapter begins with a brief description of the competition and then continues to describe our algorithmic solution to this competition which forms the core of the chapter.

Chapter 4 presents a scalability issue common to any MF model with regard to the retrieval of the recommendations (after the learning has concluded). Retrieval of recommendations requires computing a very large number of inner products between user vectors and item vectors. While this task is essential in any large scale real world recommender, the problem of efficient retrieval is hardly discussed in the academic literature. We propose two methods to efficiently retrieve recommendations in the MF framework. The core of this research was published in [5].

Chapter 5 describes a new algorithm focused on the task of item based recommendation, in which user profiles are not available or irrelevant (e.g., when the current, short-term interest is unrelated to longer term inclinations). While evidentially used in practice, we are not aware of published scientific works addressing collaborative-filtering item-oriented recommendations. We present an item-based recommendation algorithm that also embeds item vectors in a Euclidean space. Parts of this work was published in [6].

As part of my PhD, I also explored other information retrieval tasks. These earlier studies were centered on Music Information Retrieval (MIR) from Peer-to-Peer (P2P) datasets. This additional research was published in several conferences and journals [7, 8, 9, 10, 11, 12, 13] and briefly summarized in Chapter 6.

1.2 Notation

We reserve special indexing letters for distinguishing users from items: for users u and for items i, where indexes are $i \in \{1, \ldots, M\}$ for items and $u \in \{1, \ldots, N\}$ for users. We use regular lower-case fonts for denoting scalars, **bold** lower-case fonts for denoting vectors, and **bold** upper-case fonts for denoting matrices. For example, x – is a scalar, $\mathbf{x}$ – is a vector, and $\mathbf{X}$ – is a matrix. We denote by $\theta_{\mathbf{x},\mathbf{y}}$ the angle between the vectors $\mathbf{x}$ and $\mathbf{y}$ at the origin. Finally, we denote the l_2-norm of a vector $\mathbf{x}$ by $\|\mathbf{x}\|$.

II Recommenders Based on Matrix Factorization

Matrix Factorization (MF) based recommender systems model users and items by embedding both into a lower dimensional latent space in which the affinity between a user and an item is determined by the inner product of their representative vectors. In this Chapter we give an overview of MF models and learning techniques.

2.1 A Basic Model

Let us begin with a basic MF model for explicit ratings. We denote a ratings dataset $\mathcal{D} \stackrel{\text{def}}{=} \{r_{ui}\}$, where r_{ui} is a rating value given by user u to an item i. We associate each user u with a user-traits vector $\mathbf{x}_u \in \mathbb{R}^d$, and each item i with an item-traits vector $\mathbf{y}_i \in \mathbb{R}^d$, where d is the dimensionality of the latent space. The dimensionality is typically much lower than the number of users or items: $d \ll \min(N, M)$. Predicted ratings are obtained using the rule:

$$\hat{r}_{ui} = \mathbf{x}_u^\top \mathbf{y}_i. \tag{2.1}$$

Let $\mathbf{X} \in \mathbb{R}^{d \times N}$ and $\mathbf{Y} \in \mathbb{R}^{d \times M}$ be the latent user and item factor matrices with columns $\mathbf{x}_u$ and $\mathbf{y}_i$ representing the d-dimensional user-specific and item-specific latent factors of user u and item i, respectively. The predictions matrix can be written as $\hat{\mathbf{R}} = \mathbf{X}^\top \mathbf{Y}$. Hence the name *Matrix Factorization*.

It is common practice to augment the model in (2.1) with additional terms to "clean" biases and capture common patterns across users and items. We therefore extend (2.1) with the more elaborate:

$$\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{x}_u^\top \mathbf{y}_i, \tag{2.2}$$

where μ is a constant equivalent to the overall mean rating value, and b_u and b_i are scalars that represent the item and user biases, respectively.

A user bias models a user's tendency to rate on a higher or lower scale than the average rater, while the item bias captures the extent of the item popularity with regard to the overall mean rating value. We denote by $\mathbf{b}^{users} \in \mathbb{R}^N$ and $\mathbf{b}^{items} \in \mathbb{R}^M$ the vectors consisting of all the users and items biases respectively – namely, b_u is the u'th component of $\mathbf{b}^{users}$, and b_i is the i'th component of $\mathbf{b}^{items}$.

The user's trait vector $\mathbf{x}_u$ represents the user's preferences or "taste". Similarly, the item's traits $\mathbf{y}_i$ vector represent the item's latent characteristics. The dot-product $\mathbf{x}_u^\top \mathbf{y}_i$ is the personalization component, which captures user's n affinity to item m. Note that while μ is a constant that can be easily achieved by finding the mean rating, the other terms (b_u, b_i, $\mathbf{x}_u$, and $\mathbf{y}_i$) are the hidden parameters that require learning. In fact, adding μ is merely equivalent to centering the rating values.

A significant strength of MF models is their natural ability to easily augment them with more parameters to capture additional known pattern in the dataset. For example, in Chapter 3 we expand the basic model in (2.2) by including additional parameters to account for taxonomy patterns and different types of temporal dynamics.

2.2 Learning Parameters

There are various techniques for training MF models. Generally a cost function such as the Root Mean Squared Error (RMSE) is defined on the prediction error and optimization is followed by a Stochastic Gradient Descent (SGD) or an Alternating Least Squares (ALS) algorithm. ALS for MF models was introduced by Koren *et al.* in [14]. Alternatively, a Bayesian probabilistic model can be defined and inference can be done by optimizing the posterior distribution of the parameters. In Section 2.2.1 we will explain the former approach – namely, learning parameters by optimizing a squared loss cost function. In Section 2.2.2, we will discuss the alternative Bayesian approach for learning parameters, and show their equivalence in the case of Maximum A Posteriori (MAP) estimation.

2.2.1 Minimizing Squared Loss

We wish to derive a learning scheme for the model's parameters in (2.2) by optimizing a squared loss cost function. Given a dataset $\{r_{ui}\} \in \mathcal{D}$ of observed rating-tuples, our cost function C is defined as:

$$\begin{aligned} C &= \frac{1}{|\mathcal{D}|} \sum_{r_{ui} \in \mathcal{D}} (r_{ui} - \hat{r}_{ui})^2 \\ &= \frac{1}{|\mathcal{D}|} \sum_{r_{ui} \in \mathcal{D}} \left(r_{ui} - \mu - b_u - b_i - \mathbf{x}_u^\top \mathbf{y}_i\right)^2, \end{aligned} \tag{2.3}$$

where $|\mathcal{D}|$ is the number of tuples in $\mathcal{D}$.

For the set of model parameters $\Theta = (\mathbf{b}^{users}, \mathbf{b}^{item}, \mathbf{X}, \mathbf{Y})$, we learn optimal values $\hat{\Theta}$ such that

$$\begin{aligned} \hat{\Theta} &= \min_{\Theta} C \\ &= \min_{(\mathbf{b}^{users}, \mathbf{b}^{item}, \mathbf{X}, \mathbf{Y})} \frac{1}{|\mathcal{D}|} \sum_{r_{ui} \in \mathcal{D}} \left(r_{ui} - \mu - b_u - b_i - \mathbf{x}_u^\top \mathbf{y}_i\right)^2. \end{aligned} \tag{2.4}$$

We augment (2.4) with ridge regularization terms that penalize parameters' magnitude to avoid overfitting:

$$\begin{aligned} \hat{\Theta} = \min_{\Theta} &\frac{1}{|\mathcal{D}|} \sum_{r_{ui} \in \mathcal{D}} \left(r_{mn} - \mu - b_u - b_i - \mathbf{x}_u^\top \mathbf{y}_i\right)^2 \\ &+ \lambda \sum_{u,i} \left(\|b_u\|^2 + \|b_i\|^2 + \|\mathbf{x}_u\|^2 + \|\mathbf{y}_i\|^2\right), \end{aligned} \tag{2.5}$$

where λ is a regularization coefficient that can be determined using cross-validation.

It is often useful to use different regularization coefficients for different parameters (e.g., λ_{b_i} for item bias parameters, and $\lambda_{\mathbf{x}_u}$ for regularizing user trait vectors). This requires non-trivial optimization algorithms for fitting a large number of model parameters. We discuss this in Section 3.4.4.

Optimization of (2.5) is achieved either by Stochastic Gradient Descent (SGD) or Alternating Least Squares (ALS) algorithms. Generally SGD is easier and faster to con-

verge. It is also known to give slightly more accurate results [14]. When scaling is considered, ALS has an advantage since it is embarrassingly parallel.

Stochastic Gradient Descent

Optimization of (2.5) using SGD is described in Algorithm 1. The examples in the dataset $\mathcal{D}$ are iterated until convergence. At each point we update the user and item parameters of the current example by making a small step to reduce the point-wise error e_{ui}, where $e_{ui} \stackrel{def}{=} r_{ui} - \hat{r}_{ui}$. Updates are taken using a learning rate γ. Similar to the regularization coefficients, it is possible to use different update rates for different parameters. After each iteration, the learning rate is deceased using exponential decay. In Algorithm 1, the decay coefficient is 0.9. The decay coefficient and initial learning rates can be determined via cross validation as discussed Section 3.4.4.

Algorithm 1 Learning with Stochastic Gradient Descent

Input: Dataset $\mathcal{D}$
Output: Optimal parameters Θ

while not converged do
  for each $r_{ui} \in \mathcal{D}$ **do**
    $\mathbf{x}_u \leftarrow \mathbf{x}_u + \gamma\left(e_{ui}\mathbf{y}_i - \lambda\mathbf{x}_u\right)$
    $\mathbf{y}_i \leftarrow \mathbf{y}_i + \gamma\left(e_{ui}\mathbf{x}_u - \lambda\mathbf{y}_i\right)$
    $b_u \leftarrow b_u + \gamma\left(e_{ui} - \lambda b_u\right)$
    $b_i \leftarrow b_i + \gamma\left(e_{ui} - \lambda b_i\right)$
  end for
  $\gamma \leftarrow 0.9\gamma$
end while

Figure 2.1: Stochastic Gradient Descent (SGD) optimization of (2.5)**.**

Alternating Least Squares

The optimization of (2.5) is non-convex in Θ (because of the $\mathbf{x}_u^\top\mathbf{y}_i$ term). However, it is quadratic in any single parameter in Θ and thus convex in any single parameter (when the other parameters are held fixed). This naturally calls for alternating optimization

algorithms such as ALS in which we repeatedly iterate on the optimization parameters and solve for each parameter independently. Each update improves the overall objective and since our cost function is bounded (being non-negative) convergence is guaranteed.

Optimization of (2.5) using ALS is described in Algorithm 2. ALS employs different update steps that depends on several data points and require a matrix inversion operation (for the trait vectors update). For example, the user traits update step consists of a summation of all the item trait vectors belonging to the items that were rated by that user. The vectors are scaled by a factor that depends on the contribution of the other parameters, and the vectors sum is scaled by an inverted matrix that balances between the the regularization constant λ and the contribution of the vectors. The update step for user biases follows a very similar formula, and the update steps for the items' parameters are symmetrical to the users' update steps. Algorithm 2 is embarrassingly parallel - it is parallelized by updating all the user parameters in parallel and then updating all the item parameters in parallel.

2.2.2 A Bayesian Formulation

Matrix Factorization recommender systems can also give rise to forward generative Bayesian formulation [15]. Figure 2.3 depicts a graphical model representation of a model very similar to that of (2.5). Each user u is associated with a latent vector $\mathbf{x}_u \in \mathbb{R}^d$ and a user bias b_u, and each item i is associated with a latent vector $\mathbf{y}_i \in \mathbb{R}^d$ and an item bias b_i. We assume that a rating r_{ui} is determined by the model's parameters and some random noise:

$$r_{ui} = \hat{r}_{ui} + \epsilon, \tag{2.6}$$

where $\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{x}_u^\top \mathbf{y}_i$ and $\epsilon \sim \mathcal{N}(\epsilon; 0, \sigma_r^2)$. The likelihood of a single rating observation is therefore given by:

$$p(r_{ui}|\mathbf{x}_u, \mathbf{y}_i, b_u, b_i) = \mathcal{N}\left(r_{ui}; \hat{r}_{ui}, \sigma_r^2\right), \tag{2.7}$$

Algorithm 2 Learning with Alternating Least Squares

Input: Dataset $\mathcal{D}$
Output: Optimal parameters Θ

while not converged do

// Update Users:
for each $u \in \{1 \dots N\}$ **do**

$$\mathbf{x}_u \leftarrow \Big(\sum_{\text{all } i \text{ of } u} \mathbf{y}_i \mathbf{y}_i^\top + \lambda \mathbf{I} \Big)^{-1} \sum_{\text{all } i \text{ of } u} (r_{ui} - \mu - b_i - b_u)\, \mathbf{y}_i$$

$$b_u \leftarrow \Big(M_u + \lambda \Big)^{-1} \sum_{\text{all } i \text{ of } u} \left(r_{ui} - \mu - b_i - \mathbf{x}_u^\top \mathbf{y}_i \right)$$

end for

// Update Items:
for each $i \in \{1 \dots M\}$ **do**

$$\mathbf{y}_i \leftarrow \Big(\sum_{\text{all } u \text{ of } i} \mathbf{x}_u \mathbf{x}_u^\top + \lambda \mathbf{I} \Big)^{-1} \sum_{\text{all } u \text{ of } i} (r_{ui} - \mu - b_i - b_u)\, \mathbf{x}_u$$

$$b_i \leftarrow \Big(N_i + \lambda \Big)^{-1} \sum_{\text{all } u \text{ of } i} \left(r_{ui} - \mu - b_u - \mathbf{x}_u^\top \mathbf{y}_i \right)$$

end for

end while

Figure 2.2: Alternating Least Squares (ALS) optimization of (2.5)**.**

M_u **- The number of items that were rated by user** u**.**
N_i **- The number of users that rated item** i**.**

Assuming independent Gaussian priors on the parameters, we get the following posterior density:

$$\begin{aligned} p\left(\mathbf{X}, \mathbf{Y}, \mathbf{b}^{users}, \mathbf{b}^{items} | \mathcal{D}\right) \propto & \prod_{r_{ui} \in \mathcal{D}} \mathcal{N}\left(r_{ui}; \hat{r}_{ui}, \sigma_r^2\right) \\ & \prod_{u=1}^{N} \mathcal{N}\left(\mathbf{x}_u; 0, \sigma_{\mathbf{X}}^2 \mathbf{I}\right) \mathcal{N}\left(b_u; 0, \sigma_{bu}^2\right) \\ & \prod_{i=1}^{M} \mathcal{N}\left(\mathbf{y}_i; 0, \sigma_{\mathbf{Y}}^2 \mathbf{I}\right) \mathcal{N}\left(b_i; 0, \sigma_{bi}^2\right), \end{aligned} \tag{2.8}$$

and the log-posterior is:

$$\log\left(\mathbf{X}, \mathbf{Y}, \mathbf{b}^{users}, \mathbf{b}^{items} | \mathcal{D}\right) = -\frac{1}{2\sigma_r^2} \sum_{r_{ui} \in \mathcal{D}} \left(r_{ui} - \hat{r}_{ui}\right)^2 - \frac{1}{2\sigma_{\mathbf{X}}^2} \sum_u \|\mathbf{x}_u\|^2 - \frac{1}{2\sigma_{bu}^2} \sum_u b_u^2 - \frac{1}{2\sigma_{\mathbf{Y}}^2} \sum_i \|\mathbf{y}_i\|^2 - \frac{1}{2\sigma_{bi}^2} \sum_i b_i^2 + \text{const} \tag{2.9}$$

.

By defining $\lambda = \frac{\sigma_r^2}{\sigma_{\mathbf{X}}^2} = \frac{\sigma_r^2}{\sigma_{\mathbf{Y}}^2} = \frac{\sigma_r^2}{\sigma_{bu}^2} = \frac{\sigma_r^2}{\sigma_{bi}^2}$ we get complete equivalence between the model formulation of (2.9) and the model formulated in (2.5). Hence, minimization of the squared loss is equivalent to maximization of the posterior distribution of the latent parameters. From here, we can continue as before via SGD or ALS. Applying SGD is equivalent to directly optimizing log-posterior density and ALS is equivalent to the well known Expectation Maximization algorithm [16].

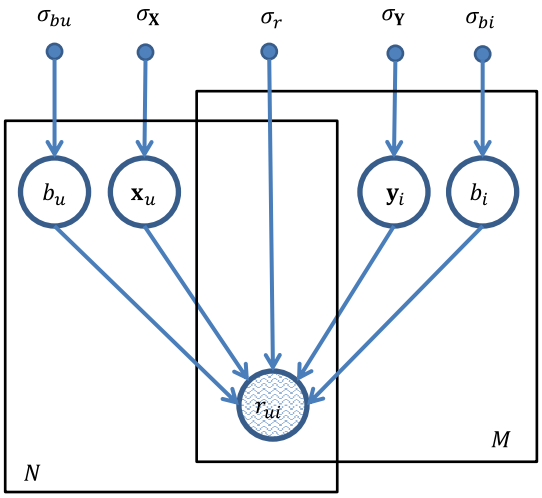


Figure 2.3: A graphical model representation of a basic recommender.

It is also possible to learn MF model using full inference methods which approximate

the posterior distribution rather than maximizing it. The formulation can be similar to the one shown here, but the learning algorithm may employ methods such as Gibbs Sampling, Variational Bayes or Expectation Propagation. In the rest of this thesis we focus on the models themselves and do not compare learning algorithm. We refer the reader to Paquet *et al.* [17] for an example of an ordinal matrix factorization implementation with Gibbs Sampling and Variational Bayes inference, and to Stern *et al.* [18] for an Expectation Propagation treatment to a MF model.

III Modeling with Temporal Dynamics and Item Taxonomy

This chapter is mostly based on research that began during my internship in Yahoo! Labs under the supervision of Yehuda Koren and Gideon Dror and continued afterwards. The work was published in [2, 3, 4].

3.1 The KDD-Cup'11 Data Challenge

This chapter is based on a dataset sampled from the Yahoo! Music database of ratings collected during 1999-2010[1] which we released as part of the KDD-Cup'11 challenge [2]. The challenge attracted thousands of participants and after the contest concluded the dataset itself was adopted by many researchers for a variety of studies. The contest offered two different tasks. The first track – Track1 focused on predicting users' ratings. The evaluation criterion was the Root Mean Squared Error (RMSE) between predicted ratings and true ones. Track1 was similar in nature to the well known Netflix competition [19] with the added complexity of items taxonomies and fine grained temporal resolution.

As organizers, we did not participate in the actual contest, but the model described in this Chapter was designed under the same settings as the Track1 challenge. The best result achieved by the winners – "National Taiwan University" was RMSE=21 [20]. The model described here achieved RMSE=22.59 (see Section 3.5). However, there is a significant difference between our solution and the solutions of the top participants: All the top contest solutions employed ensembles that blend predictions from many recommendation models. Blending solutions from a large number of simple predictors is a well known method to reduce generalization error. We on the other hand, focused on building one single model that will introduce new techniques for recommender systems. Specifically we focused on modeling taxonomies and different temporal dynamics. Nevertheless, we

[1] Publicly available at `http://webscope.sandbox.yahoo.com/catalog.php?datatype=c`

were happy to discover that the model's performance were competitive throughout the competition – even as a single model, the final result achieved by our solution would have been ranked on the 10th position of about 2100 *active* participants at the final stage of the competition.

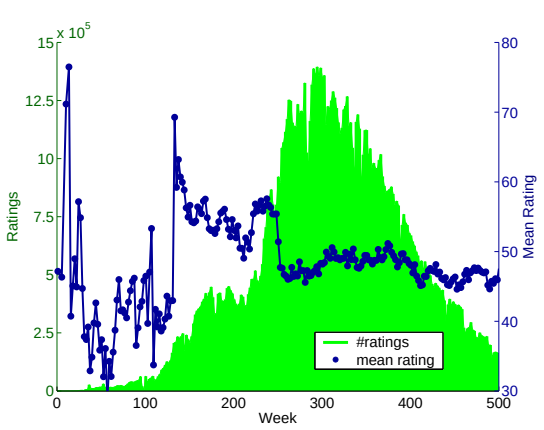
The second track – Track2 focused on a classification task: For each user in Track2 test-set six items were listed. Three out of these six items have never been rated by that user, whereas the other three items were rated "highly" by the user, that is, a score of 80 or higher. The goal of Track2 was to differentiate high ratings from missing ones. The evaluation criterion was the error rate – the fraction of wrong predictions. In the remaining of this chapter, we will focus on Track1. The reader is referred to [2] for more information about Track2.

3.2 The Yahoo! Music Dataset

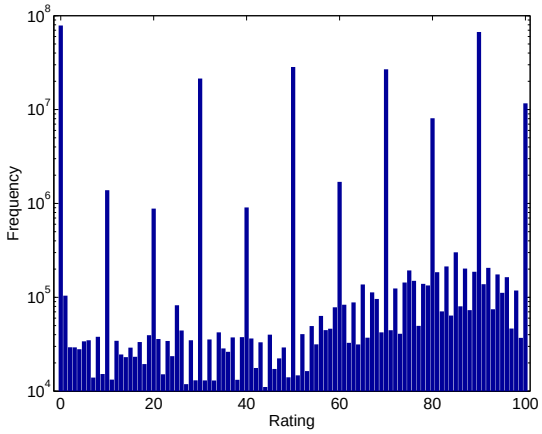
The Yahoo! Music dataset comprises of 262,810,175 ratings of 624,961 music items by 1,000,990 users. The ratings include one-minute resolution timestamps, allowing refined temporal analysis. Each item and each user has at least 20 ratings. The available ratings were split into train, validation and test sets, such that the last 6 ratings of each user were placed in the test set and the preceding 4 ratings were used in the validation set. The train set consists of all earlier ratings (at least 10). The total sizes of the train, validation and test sets were therefore 252,800,275, 4,003,960, and 6,005,940, respectively. Next, we discuss the descriptive statistics of this dataset which motivate our model.

3.2.1 Characterizing the Yahoo! Music Dataset

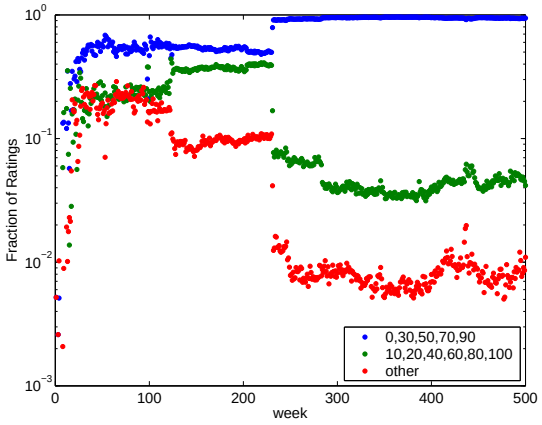
Figure 3.1(a) depicts the weekly number of ratings and the weekly mean ratings score vs. the number of weeks that passed since the launch of the service in 1999. The ratings are integers between 0 and 100. Figure 3.1(b) depicts the distribution of ratings in the train set using a logarithmic vertical scale. The vast majority of the ratings are multiples of ten, and only a minuscule fraction are not. This mixture reflects the fact that several interfaces ("widgets") were used to rate the items, and different users had different rating



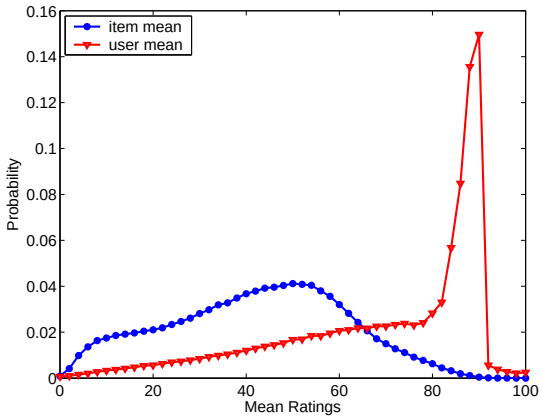
(a) Number of ratings and mean rating score vs. time.



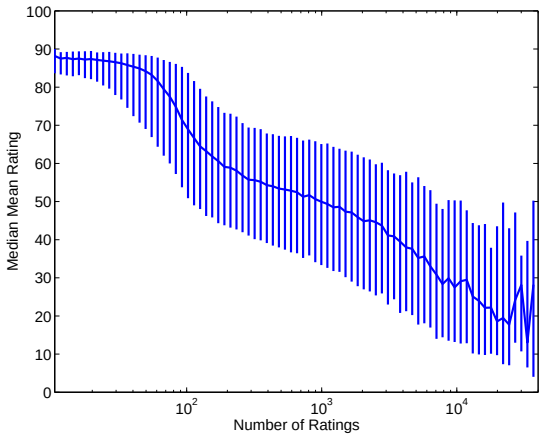
(b) Rating-values histogram



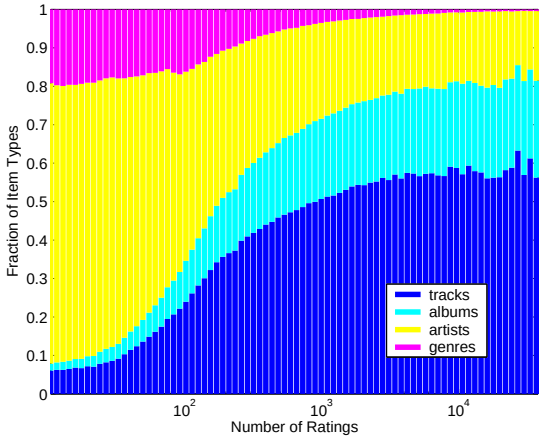
(c) Relative frequency of the three groups of ratings as a function of time.



(d) The distributions of item and user mean ratings.



(e) Median of user ratings as a function of the number of ratings issued by the user. The vertical lines represent interquartile range.



(f) The fraction of ratings the four item types receive as a function of the number of ratings a user gives.

Figure 3.1: The Yahoo! Music Dataset Statistics

"strategies". The popularity of a widget used to enter ratings at a 1-to-5 star scale is reflected by the dominance of the peaks at $0, 30, 50, 70$ and 90 into which star ratings

were translated.

An interesting aspect of the data is the fact that widgets have been altered throughout the years. Figure 3.1(c) depicts the relative frequency of each of the three types of ratings. In the first group are the ratings corresponding to the five dominant peaks of Figure 3.1(b) $(0, 30, 50, 70$ and $90)$. The second group includes the remaining peaks $(10, 20, 40, 60, 80$ and $100)$, and the third group contains the remaining ratings (those not divisible by 10). Abrupt changes in the relative frequencies of the three groups are clearly observed on the 125-th week as well as on the 225-th week. These dates are also associated with a dramatic change in mean rating, as can be observed in Figure 3.1(a).

We calculated the mean rating of each user, as well as the mean rating of each item. Figure 3.1(d) depicts these two distributions. The location of the modes (at 89 and 50 respectively), as well as the variances of the two distributions are quite distinct. In addition, the distribution of the mean user ratings is significantly more skewed.

Different rating behavior of users accounts for the apparent difference between the distributions. It turns out that users who rate more items tend to have considerably lower mean ratings. Figure 3.1(e) substantiates this effect. Users were binned according to the number of items they rated, on a linear scale. The graph shows the median of the mean ratings in each bin, as well as the interquartile range in each bin plotted as a vertical line. One of the explanations for this effect is that "heavy" raters, those who explore and rate tens of thousands of items, tend to rate more items that do not match their own musical taste and preferences, and thus the rating scores tend to be lower.

A distinctive feature of this dataset is that user ratings are given to entities of four different types: *tracks, albums, artists, and genres.* The majority of items (81.15%) are tracks, followed by albums (14.23%), artists (4.46%) and genres (0.16%). The ratings however, are not uniformly distributed: Only 46.85% of the ratings belong to tracks, followed by 28.84% to artists, 19.01% to albums and 5.3% to genres. Moreover, these proportions are strongly dependent on the number of ratings a user has entered. Heavier raters naturally cover more of the numerous tracks, while the light raters mostly concentrate on artists; the effect is shown in Figure 3.1(f). Thus, unlike the train set,

the validation and test sets, which equally weight all users, are dominated by the many light-raters and dedicate most of their ratings to artists rather than to tracks.

3.2.2 Items Taxonomy

All rated items are tied together within a taxonomy. That is, for a track we know the identity of its album, performing artist and associated genres. Similarly we have artist and genre annotation for the albums. There is no genre information for artists, as artists may switch between many genres in their career. We denote by $album(i)$ and $artist(i)$ the album and the artist of track i respectively. Similarly, for albums, we denote by $artist(i)$ the artist of album i. Tracks and albums in the Yahoo! Music dataset may belong to one or more genres. We denote by $genres(i)$ the set of genres of item i. Lastly, we denote by $type(i)$ the type of item i, with $type(i) \in \{track, album, artist, genre\}$. We show that this taxonomy is particularly useful, due to the large number of items and the sparseness of data per item (mostly attributed to "tracks" and "albums").

Although this model focuses on the Yahoo! Music dataset, many of the characteristics described here are not limited to this dataset. A hierarchy of item categories, for example, is a very common feature relevant to most web recommender systems. Indeed, e-commerce sites, e.g. those selling books, movies or electronics, tend to arrange their items within a taxonomy. Our experience is that other recommender system datasets (like the Netflix dataset [19]) also exhibit effects similar to those presented here. For example, a non-stationary distribution of ratings as depicted in Figure 3.1(a) is a common phenomenon, which usually stems from the change in the supply of items, from upgrades of web interfaces or from social, political or economic trends affecting the taste and inclination of the users; see, e.g., [21]. On the other hand, we are not aware of other recommender system datasets where items attached to different levels of the taxonomy can be rated. While this phenomenon is more relevant to music datasets, one could reasonably imagine movie recommenders asking users to rate full genres, or book recommenders asking users to rate book authors.

3.3 Modeling Bias Patters

In the context of rating systems, biases model the portion of the observed signal that is derived either solely by the user or solely by the rated item, but not by their interaction. For example, a user bias may model a user's tendency to rate higher or lower than the average rater, while an item bias may capture the extent of the item's popularity. Figure 3.1(d) illustrates the fact that the mean ratings of various users and items are quite diverse, suggesting the importance of modeling these two types of biases.

3.3.1 The Need for Biases

Since components of the user bias are independent of the item being rated, while components in the item bias are independent of any user, they do not take part in modeling personalization, e.g., modeling user musical taste. After all, ordering items by using biases only necessarily produces the same ranking for all users, hence personalization—the cornerstone of recommendation systems—is not achieved at all.

Lack of personalization power should not be confused with lack of importance for biases. There is plenty of evidence that much of the observed variability in rating signal should be attributed to biases. Hence, properly modeling biases would effectively amount to cleaning the data from signals unrelated to personalization purposes. This will allow the personalization part of the model (e.g., matrix factorization), where users and items do interact, to be applied to a signal more purely relevant for personalization. Perhaps the best evidence is the heavily analyzed Netflix Prize dataset [19]. The total variance of the ratings in this dataset is 1.276, corresponding to a Root Mean Squared Error (RMSE) of 1.1296 by a constant predictor. Three years of multi-team concentrated efforts reduced the RMSE to 0.8556, thereby leaving the unexplained ratings variance at 0.732. Hence the fraction of explained variance (known as R^2) is 42.6%, whereas the rest 57.4% of the ratings variability is due to unmodeled effects (e.g., noise). Now, let us analyze how much of the explained variance should be attributed to biases, unrelated to personalization. The best published pure bias model [22] yields an RMSE=0.9278, which is equivalent to reducing the variance to 0.861 thereby explaining 32.5% of the observed variance.

This (quite surprisingly) means that the vast majority of the 42.6% explainable variance in the Netflix dataset, should be attributed to user and item biases having nothing to do with personalization. Only about 10% of the observed rating variance comes from effects genuinely related to personalization. In fact, as we will see later (Section 3.5), our experience with the music dataset similarly indicates the importance role biases play. Here the total variance of the test dataset is 1084.5 (reflecting the 0-100 rating scale). Our best model could reduce this variance to around 510.3 ($R^2 = 52.9\%$). Out of this 52.9% explained variance, once again the vast majority (41.4%) is attributed to pure biases, leaving about 11.5% to be explained by personalization effects. Hence, the big importance one should put on well modeling biases.

We present a rich model for both the item and user biases, which accounts for the item taxonomy, user rating sessions, and items' temporal dynamics. In the following we will gradually develop the basic user and item biases into a a rich model that accounts for a variety of patterns in the dataset.

3.3.2 A Basic Bias Model

The most basic bias model captures the main effects associated with users and items [23]. We start with the basic biases of (2.2). In order to focus on the biases and measure their contribution, we isolate the biases by removing the user and item trait vectors (the personalization component). This gives rise to the model

$$b_{ui} = \mu + b_u + b_i \tag{3.1}$$

Since components of the user bias are independent of the item being rated, while components in the item bias are independent of any user, they do not take part in modeling personalization, e.g., modeling the musical taste of a user. After all, ordering items by using only a bias model (3.1) necessarily produces the same ranking for all users, hence personalization—the cornerstone of recommendation systems—is not achieved at all. Yet, there is plenty of evidence that much of the observed variability of ratings is attributed to biases. Hence, properly modeling biases would effectively amount to

cleaning the data from patterns unrelated to personalization purposes. This will allow the personalization part of the model (e.g., matrix factorization) to be applied to signals much more relevant to personalization, where users and items do interact.

We extend the model (3.1) and present a rich model for both the item and user biases, which accounts for the item taxonomy, user rating sessions, and items' temporal dynamics. We will gradually add components to the user and item biases to capture additional patterns. Let us denote by b_u^0 and b_i^0 our initial (stage 0) user and item biases respectively. Namely b_u^0 is simply b_u and b_i^0 is simply b_i.

3.3.3 Taxonomy biases

We start by letting item biases share components for items linked by the taxonomy. For example, tracks in a good album may all be rated somewhat higher than the average, or a popular artist may have all her songs rated a bit higher than the average. We therefore add shared bias parameters to different items with a common ancestor in the taxonomy hierarchy. We expand the item bias model for tracks as follows

$$b_i^1 = b_i + b_{album(i)} + b_{artist(i)} + \frac{1}{|genres(i)|} \sum_{g \in genres(i)} b_g, \tag{3.2}$$

where b_i^1 is our new item bias. Here, the total bias associated with a track i sums both its own specific bias modifier (b_i), together with the bias associated with its album ($album(i)$) and its artist ($artist(i)$), and the mean bias associated with its genres($\frac{1}{|genres|} \sum_{g \in G} b_g$).

Similarly for each album we expand the bias model as follows

$$b_i^1 = b_i + b_{artist(i)} + \frac{1}{|genres(i)|} \sum_{g \in genres(i)} b_g \tag{3.3}$$

One could view these extensions as a gradual accumulation of the biases. For example, when modeling the bias of album i, the start point is $b_{artist(i)} + \frac{1}{|genres(i)|} \sum_{g \in genres(i)} b_g$, and then b_i adds a residual correction on top of this start point. Similarly, when i is a track another track-specific correction is added on top of the above. As bias estimates for tracks and albums are less reliable, such a gradual estimation allows basing them on

more robust initial values. Note that such a framework not only relates items to their taxonomy ancestors, but (indirectly) also to other related items in the taxonomy. For example, a track will get related to all other tracks in its album, and to lesser extent to all other tracks by the same artist. Also, note that while artists and genres are less susceptible to the sparsity problem, they also benefit from this model as ratings to tracks and albums also influence the biases of their corresponding artist and genre.

The taxonomy of items is also useful for expanding the user bias model. For example, a user may tend to rate artists or genres higher than songs. Therefore, given an item i the user bias is

$$b_u^1 = b_u + b_{u,type(i)} \tag{3.4}$$

where b_u is the user specific bias component and $b_{u,type(i)}$ is a shared component of all the ratings by user u to items of type $type(i)$.

3.3.4 User Activity Sessions

Ratings are marked by a date and and timestamp with resolution down to minutes. We used this information for modeling temporal dynamics of both items and users. We start by modeling user listening sessions. It is common for users to listen to many songs and rate them one after the other. A rating session is therefore a set of consecutive ratings without an extended time gap between them. In our implementation user sessions are separated by at least 5 hours of idle gap (no rating activity). There are many psychological phenomena that affect ratings grouped in a single session. These effects are captured by user session biases.

One example is the fact that the order in which the songs were listened by the user might determine the ratings scores, a phenomenon known as the drifting effect [24]. Users tend to rate items in the context of previous items they rated. If the first song a user hears is particularly good, the following items are likely to be rated by that user lower than the first song. Similarly, if the user did not enjoy the first song, the ratings of subsequent songs may shift upwards. The first song therefore may serve as a reference rating to all the following ratings. However, with no absolute reference for the first rating, different

users begin rating at different scales and some users tend to give it a default rating (e.g., 70 or 50). Consequently, all the following ratings in that same session may be biased higher or lower according to the first rating. Another source for session biases is the mood of the user. A user may be in a good/bad mood that may affect her ratings within a particular session. It is also common to listen to similar songs in the same session, and thus their ratings become similar.

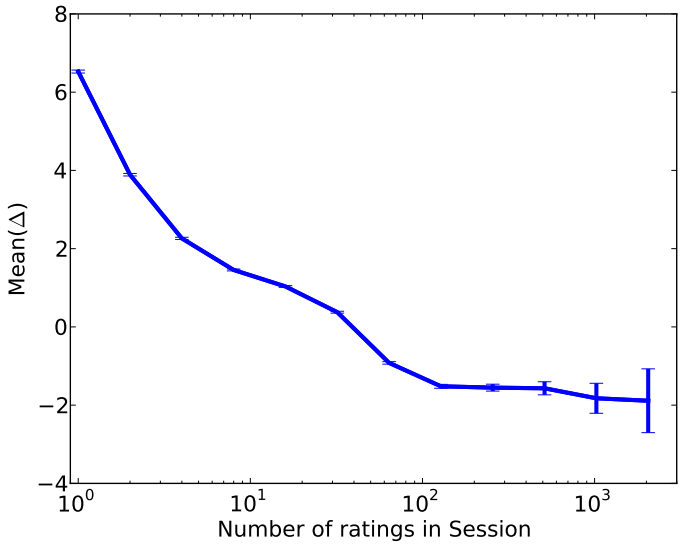


Figure 3.2: The difference between the mean rating of a session to the mean rating of the corresponding user as a function of session length, averaged over all sessions

The Yahoo! Music dataset exhibits another form of session bias, where longer sessions tend to have a lower mean rating. This is not surprising, given the lower average rating for "heavier" users, as summarized in Figure 3.1(e). But more importantly, this claim is correct even on a per-user basis, namely for each user longer sessions tend to have a significantly lower mean rating. To show this we calculated for each session the difference, Δ, between the mean rating of the session to the mean rating of the corresponding user. Averaging Δ over all sessions and plotting it as a function of the sessions' length results in Figure 3.2. The error-bars represent a 0.95 confidence interval of the estimate of the mean value of Δ for each length. The latter was binned on a logarithmic scale. The figure shows that long sessions comprising 100 ratings or more are on average about 2 points lower than the mean rating of the user, whereas the shortest sessions, comprising

a single rating, are on average 6 points higher than the mean rating of the user.

To take such effects into account, we added a session bias term to our user bias model. We denote by $session(u, i)$ the rating session of the rating r_{ui}, and expand our user bias model to include session biases

$$b_u^2 = b_u^1 + b_{u,session(i,u)} \tag{3.5}$$

The session bias parameter $b_{u,session(i,u)}$ models the bias component common to all ratings of u in the same session she rated i.

3.3.5 Item Temporal Biases

The popularity of songs may change dramatically over time. While users' temporal dynamics seem to follow abrupt changes across sessions, items' temporal dynamics are much smoother and slower, thus calling for a different modeling approach. We follow here an approach suggested by Piotte and Chabbert [25].

Given item i and the time t since i's first rating, we define a time dependent item bias as a linear combination of n temporal basis functions $\mathbf{f}(t) = (f_1(t), f_2(t), ...f_n(t))^\top$ and expand the item bias component to be

$$b_i^2 = b_i^1 + \mathbf{c}_i^\top \mathbf{f}(t), \tag{3.6}$$

where $\mathbf{c}_i \in \mathbb{R}^n$ is an item specific vector of coefficients.

We learned both $\mathbf{f}(t)$ and $\mathbf{c}_i$ using SGD minimization of a squared loss cost function in a similar fashion to all other model components; see Section 3.4.3. In practice, a 2-week coarse time resolution is sufficient for the rather slow changing item temporal dynamics, therefore the basis functions are only estimated at a small number of points and can be easily learned. This process does not guarantee an orthogonal or normalized basis, however it finds a basis that fits the patterns seen in the dataset.

We have found that a basis of 4 functions is sufficient to represent the temporal dynamics of item biases in the Yahoo! Music dataset. Figure 3.3 depicts the learned basis

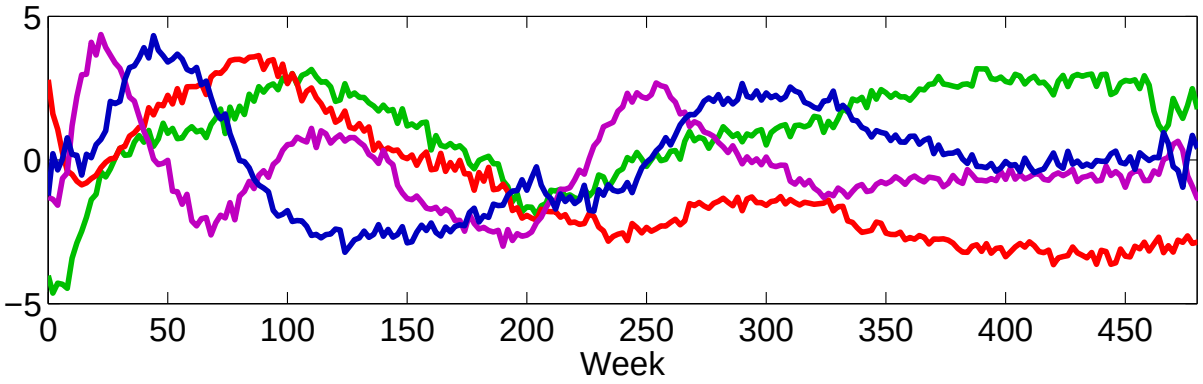


Figure 3.3: Items temporal basis functions $\{f_i(t)\}_{i=1}^4$ vs. time since an item's first rating measured in weeks

functions $\{f_i(t)\}_{i=1}^4$. Since the items' coefficients can be be either positive or negative, it is hard to give a clear interpretation to any specific basis function. However, an interesting observation is that basis functions seem to have high gradients (fluctuations) right after an item was released, indicating more dynamic temporal effects in this time period. It is also interesting to note that after a long time period (above 360 weeks), the temporal basis functions converge into relatively steady values. This indicates that at a longer perspective, items seem to have either a positive or a negative bias, with much less temporal dynamics.

3.3.6 A Full Bias Model

To summarize, our complete bias model, including both enhanced user and item biases is (for a track i)

$$\begin{aligned} b_{ui} =& \mu + b_{u,type(i)} + b_{u,session(i,u)} + b_i + b_{album(i)} \\ &+ b_{artist(i)} + \frac{1}{|genres(i)|} \sum_{g \in genres(i)} b_g + \mathbf{c}_i^\top \mathbf{f}(t_{ui}) \end{aligned} \tag{3.7}$$

where t_{ui} is the time elapsed from i's first rating till u's rating of i.

Learning the biases is performed together with the other model components by SGD as described in Section 3.4.3. The extended bias model dramatically reduced the RMSE even before any personalization components were added into the model (see results in

Section 3.5). Biases were able to absorb much of the effects irrelevant to personalization. Such a "cleaning" proved to be a key for accurately modeling personalization in later stages.

3.4 Modeling Personalization

We start with a basic personalization component similar to (2.2): Each user u is associated with a user-factor vector $\mathbf{x}_u \in \mathbb{R}^d$, and each item i with an item-factor vector $\mathbf{y}_i \in \mathbb{R}^d$. Predictions are done using the rule

$$\hat{r}_{ui} = b_{ui} + \mathbf{x}_u^\top \mathbf{y}_i \tag{3.8}$$

where b_{ui} is the bias model (3.7), and $\mathbf{x}_u^\top \mathbf{y}_i$ is the personalization model which captures user's u affinity to item i. In the following section we expand this basic personalization model to encompass more patterns observed in the data.

3.4.1 Utilizing Taxonomy

Musical artists often have a distinct style that can be recognized in all their songs. Similarly, artists style can be recognized across different albums of the same artist. Therefore, we introduce shared factor components to reflect the affinity of items linked by the taxonomy. Specifically, for each artist and album, we employ a factor vector $\mathbf{v}_i \in \mathbb{R}^d$ (in addition to also using the aforementioned $\mathbf{y}_i$). We expand our item representation for tracks to explicitly tie tracks linked by the taxonomy

$$\tilde{\mathbf{y}}_i \stackrel{\text{def}}{=} \mathbf{y}_i + \mathbf{v}_{album(i)} + \mathbf{v}_{artist(i)} \tag{3.9}$$

Therefore, $\mathbf{y}_i$ represents the difference of a specific track from the common representation of all other related tracks, which is especially beneficial when dealing with less popular items.

Similarly, we expand our item representation for albums to be

$$\tilde{\mathbf{y}}_i \stackrel{\text{def}}{=} \mathbf{y}_i + \mathbf{v}_{artist(i)} \tag{3.10}$$

One may also add shared factor parameters for tracks and albums sharing the same genre, similarly to the way genres were exploited for enhancing biases. However, our experiments did not show an RMSE improvement by incorporating shared genre information. This indicates that after exploiting the shared information in albums and artists, the remaining information shared by common items of the same genre is limited.

3.4.2 Personalizing Listening Sessions

As discussed earlier, much of the observed changes in user behavior are local to a session and unrelated to longer term trends. Thus, after obtaining a fully trained model (hereinafter, "Phase I") we perform a second phase of training, which isolates rating components attributed to session-limited phenomena. In this second phase, when we reach each user session, we try to absorb any session specific signal in separated component of the user factor. To this end we expand the user representation into

$$\tilde{\mathbf{x}}_u = \mathbf{x}_u + \mathbf{x}_{u,session} \tag{3.11}$$

where the user representation $\tilde{\mathbf{x}}_u$ consists of both the original user factor $\mathbf{x}_u$ and and the session factor vector $\mathbf{x}_{u,session}$. We learn $\mathbf{x}_{u,session}$ by fixing all other parameters and making a few (e.g., 3) SGD iterations only on the ratings given in the current session in order to learn $\mathbf{x}_{u,session}$. After these iterations, we are able to absorb much of the temporary per-session user behavior into $\mathbf{x}_{u,session}$, which is not explained by the model learned in Phase I. We then move to a final relaxation step, where we run one more iteration over all ratings in the same session, now allowing all other model parameters to change and shed away any per session specific characteristics. Since $\mathbf{x}_{u,session}$ already captures much of the per-session effects of the user factor, the other model parameters adjust themselves accordingly and capture possible small changes since the previous rating

session. After this relaxation step, we reset $\mathbf{x}_{u,session}$ to zero, and move on to the next session, repeating the above process.

Our approach is related to [21], which has also employed day specific factor vectors for each user. However, there are two notable differences. First, we apply a more refined session analysis rather than working at a coarser day resolution. Second, we employ a much more memory efficient method: since we discard the per session components $\mathbf{x}_{u,session}$ after iterating through each session, there is no need to store session vectors for every session in the dataset. At the time of prediction, we only use the last session vector. We therefore avoid the high memory consumption that occurs in previous approaches. For example, we identified 13,844,810 ratings sessions in the Yahoo! Music dataset (for all users). Using a 100-D factorization model with single precision floating point numbers (4 bytes), it would have taken more than 5.5GB of memory to store all the user session factors, significantly larger than the 400MB required to store only a single session factor for each user.

3.4.3 Stochastic Gradient Descent (SGD) Optimization

Our final prediction model takes the following form

$$\hat{r}_{ui} = b_{ui} + \tilde{\mathbf{x}}_u^\top \tilde{\mathbf{y}}_i \tag{3.12}$$

where b_{ui} is the detailed bias model as in (3.7), $\tilde{\mathbf{y}}_i$ is our enhanced item factor representation as described in (3.9) and (3.10), and $\tilde{\mathbf{x}}_u$ is defined in (3.11).

As previously alluded, learning proceeds by stochastic gradient descent (SGD), where all learned parameters are regularized with ridge (l_2) regularization. SGD visits the training examples one-by-one, and for each example updates its corresponding model parameters. The update steps are derived in a similar fashion to the update steps in Section 2.2. More specifically, for training example (u, i), SGD lowers the squared prediction error $e_{ui}^2 = (r_{ui} - \hat{r}_{ui})^2$ by updating each individual parameter θ by

$$\Delta\theta = -\gamma \frac{\partial e_{ui}^2}{\partial \theta} - \lambda\theta = 2\eta e_{ui} \frac{\partial \hat{r}_{ui}}{\partial \theta} - \lambda\theta, \tag{3.13}$$

where γ is the learning rate and λ is the regularization rate.

The Yahoo! Music dataset spans over a very long time period (a decade). In such a long period musical taste of users slowly drifts. We therefore expect model parameters to change with time. We exploit the fact that the SGD optimization procedure gradually updates the model parameters while visiting training examples one by one. It is a common practice in online learning to order training examples by their time, so when the model training is complete, the learned parameters reflect the latest time point, which is most relevant to the test period. Since we perform a batch learning including several sweeps through the dataset, we need to enhance this simple technique.

We loop through the data in a cyclic manner: we visit user-by-user, whereas for each user first we *sweep forward* from the earliest rating to the latest one, and then (after also visiting all other users) we *sweep backward* from the latest rating to the earliest one, and so on. This way, we avoid the otherwise discontinuous jump from the latest rating to the first one when starting a new iteration. This allows parameters to slowly drift with time as the user changes her taste. The process always terminates with a forward iteration ending at the latest rating.

3.4.4 Tuning Meta-parameters with The Nelder-Mead Algorithm

For each type of learned parameter we set a distinct learning rate (aka, step size) and regularization rate (aka, weight decay). This grants us the flexibility to tune learning rates such that, e.g., parameters that appear more often in a model are learned more slowly (and thus more accurately). Similarly, the various regularization coefficients allow assuming different scales for different types of parameters.

We have used the validation dataset to find proper values for these meta-parameters. Optimization of meta-parameters is a costly procedure, since we know very little on the behavior of the objective function, and because every evaluation requires running the SGD algorithm on the entire dataset. The fact that we have multiple learning and regularization parameters further complicates the matter. For optimizing more than 20 meta-parameters we resorted to the Nelder-Mead simplex search algorithm [26]. Though

not guaranteed to converge to the global minimum [27], Nelder-Mead search is a widely used algorithm with excellent results on real world scenarios [25, 28]. To speed up the search we implemented a parallel version of the algorithm as in [29]. We consider such an automated meta-parameters optimization process as key ingredient in enabling the development of a rich and flexible model.

3.5 Evaluation

We learned our model on the train dataset using SGD with 20 iterations. We used the validation dataset for early termination and for setting model-parameters; see Section 3.4.4. We then tested the results in terms of RMSE as described here.

We measured the RMSE of our predictions as we gradually add components to the bias models, and then as we gradually add components to the personalization model. This approach allows isolating the contribution of each component in the model. The results are presented in Table 3.1.

The most basic model is a constant predictor. In the case of the RMSE cost function, the optimal constant predictor would be the mean train rating, $\hat{r}_{ui} = \mu$; see row 1 of the table. In row 2 we present the basic bias model $\hat{r}_{ui} = \mu + b_i + b_u$ (3.1) (b_u^0 and b_i^0). In row 3 we report the results after expanding the item and user biases to include also taxonomy terms (b_u^1 and b_i^1), which mitigate data sparseness by capturing relations between items of the same taxonomy; see Section 3.3.3. We then added the user session bias of Section 3.3.4 (b_u^2). This gave a significant reduction in terms of RMSE as reported in row 4. We believe that modeling session biases in users' ratings is key in explaining ratings behavior in domains like music in which users evaluate and rate multiple items at short time frames. In row 5 we add the item temporal bias from Section 3.3.5 (b_i^2). This term captures changes in item biases that occur over the lifespan of items since their first ratings. This bias is especially useful in domains in which item popularity easily changes over time such as in music, or datasets in which the ratings history is long. The result in row 5 reflects the RMSE of our final bias model (defined in Section 3.3.6), when no personalization is yet in place.

#	Model Name	RMSE
1	Mean Score	38.0617
2	Items and Users Bias	26.8561
3	Taxonomy Bias	26.2553
4	User Sessions Bias	25.3901
5	Items Temporal Dynamics Bias	25.2095
6	MF	22.9533
7	Taxonomy	22.7906
8	Final	22.5918

Table 3.1: Root Mean Squared Error (RMSE) of the evolving model. RMSE reduces while adding model components.

We move on to personalized models, which utilize a matrix factorization component with 50 dimensions. The model of (3.8) yields RMSE of 22.9235 (row 6). By adding taxonomy terms to the item factors, we were able to reduce this result to 22.8254 (row 7). Finally, in row 8 we report the full prediction model including user session factors (as in Section 3.4.2). The relatively large drop in RMSE, even when the model is already fully developed, highlights the significance of temporal dynamics at the user factor level.

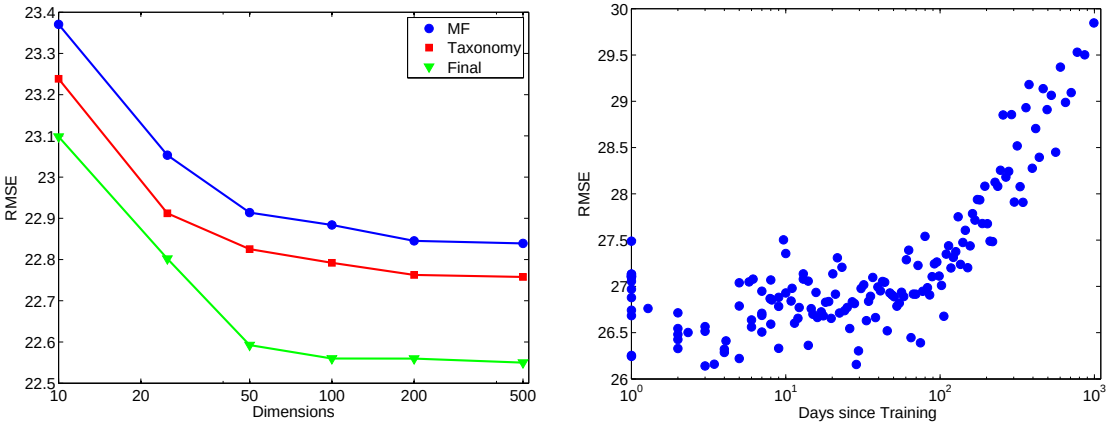
Let us consider the effect of the taxonomy on the RMSE results of each item type. Table 3.2 breaks down the RMSE results per item type of the three personalized models. It is clear that incorporating the taxonomy is most helpful for the sparsely rated tracks and albums. It is much less helpful for artists, and becomes counter-productive for the densely rated genres.

We further investigate the performance of our model as more factors are added. Figure 3.4(b) depicts the RMSE vs. factors dimensionality. Since we carefully tune the regularization terms, there is no overfitting even as the dimensionality reaches 500. However, there are clearly diminishing returns of increasing dimensionality, with almost no improvement over 100 dimensions. Note that the reduction in RMSE given by the taxonomy and session factors remains steady even as the dimensionality increases.

Lastly, we investigate the relation of the test RMSE to the time distance from the train set. Figure 3.4(b) depicts the mean RMSE and the elapsed time for each bin. Ratings with a zero elapsed time, which mostly correspond to user sessions artificially split between train and test set, were excluded from this analysis, as they are not relevant

	Track	Album	Artist	Genre
%Test	28.7%	11.01%	51.61%	8.68%
MF	27.1668	24.5203	20.9815	15.7887
Taxonomy	26.8899	24.3531	20.8766	15.7965
Final	26.85	24.1854	20.566	15.4801

Table 3.2: RMSE per item type for the three personalized models. We also report the fraction of each item type in the test dataset.



(a) RMSE vs. dimensionality of factors (d). We track regular MF, MF enhanced by taxonomy and the final model.

(b) RMSE vs. number of days elapsed since last user's training example.

Figure 3.4: Breaking down RMSE of the final model

for any real recommender system. The plateau on the left part of the figure suggests that the performance of the model is stable for about three months since the time it was trained, whereupon the RMSE of its predictions gradually increases. Thus the model needs updating only once in a month or so in order to exhibit uniform performance.

3.5.1 Visualizing the Latent Space

Figure 3.5 depicts the item factor vectors for some of the more popular artists and genres based on a 2-D model ($d = 2$). We see that most items are located within a 90° sector. At the right hand side, we see a concentration of Hip-Hop, Rap and R&B artists and genres, while various Rock groups and sub-genres of Rock are located on the left hand side of the sector. The fact that the two groups are orthogonal, means that there is little correlation between people who listen to Hip-Hop and people who listen to Rock. In other words, a user's tendency towards Hip-Hop does not convey any information about

her taste in Rock. These two genres (Hip-Hop and Rock) are the most popular genres in our dataset. Therefore, a 2-D system places them as the main two extremes. Figure 3.5 emphasizes the need for more than 2 dimensions when modeling music preferences. Since the two dimensions represent Rock and Hip-Hop, there are no dimensions left for other genres. Therefore they are constrained to lie between these two groups. We thus see a concentration of Pop genres and artists at an angle that is between Rock and Hip-Hop. Similarly genres such as Jazz, Electronic/Dance, Latin and an artist such as Bob Marley (Reggae) are placed somewhere between Hip-Hop and Rock. In higher dimensional factorization models, all these items become mostly orthogonal to each another.

The norm of an item vector determines how suitable that item would be to a user in the same direction. For example, the vector for the Industrial Rock group Nine Inch Nails is at an angle between Nirvana and Metallica, but with a much higher magnitude. However, the latter two have much larger item biases. This means that while Nirvana and Metallica tend to get higher ratings in general (as indicated by their higher item biases), Nine Inch Nails will get a higher inner product values with user vectors in its direction. Therefore, the ratings for Nine Inch Nails are more dependent on a user's musical taste (the personalization component), whereas Metallica and Nirvana are more popular in general.

As we noted above, most of the items are placed within a 90° sector. We further verified this by computing the inner product between all possible item pairs when using 50-D factor vectors and found that 96.36% of them were nonnegative. Hence, negative correlations between items ("users who like an item dislike another one") are rare. This indicates that in music, personalization is mostly based on modeling patterns of positive correlations between items. The prominence of this finding prompted us to compare it to other domains. We repeated the same experiment using the Netflix movies dataset [19]. We have found that negative correlations between movies are more likely than in music. While the majority of inner products between movie factor vectors are nonnegative, a non-negligible fraction (32.35%) is of negative inner products.

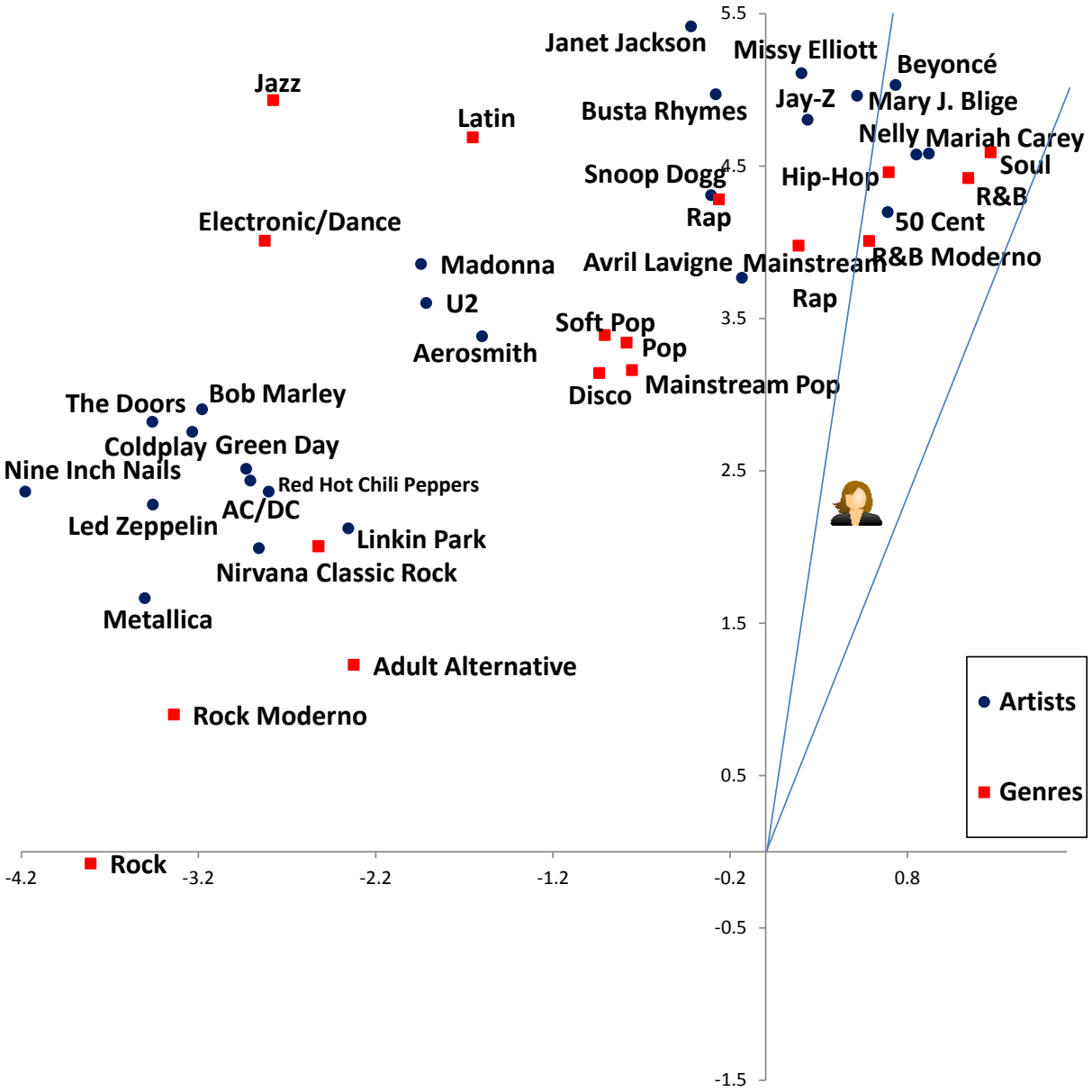


Figure 3.5: The most popular musical tracks and genres in the Yahoo! Music dataset are embedded into a 2-dimensional. The open cone suggests the region of highly preferred items for the user (her logo is at the center of the cone). Note how the learned embedding separates Rock and similar items from Hip-Hop and similar items. The low dimensionality (which is required for a visualization), causes a small number of items to be wrongly folded near less related items (e.g., Bob Marley).

IV Efficient Retrieval of Recommendations

Real world large scale MF models run into a difficulty rarely discussed in the academic literature – the computational cost of finding the top-rated items to recommend. This task involves finding the maximum dot-product for a given user vector $\mathbf{x}_u$ over a set of items S such that:

$$\mathbf{x}_u^\top \mathbf{y}_i = \max_{\mathbf{y}_i \in S} \mathbf{x}_u^\top \mathbf{y}_i. \tag{4.1}$$

Surprisingly, we did not find any technique to efficiently solve this problem; a linear search over the set of points appears to be the state-of-the-art. Moreover, the number of users is usually very high (typically higher than the number of items) and each user requires a new search over the items. A very simplistic visualization of this task is depicted in Figure 3.5 from Chapter 3. For the given user, the best recommendations (in this case songs) lie within the open cone around the user vector (maximizing the $\cos(\theta_{\mathbf{x}_u,\mathbf{y}_i})$ term) and are as far as possible from the origin (maximizing the $\|q_i\|$ term).

The problem of efficient Retrieval of Recommendations (RoR) in collaborative filtering has been previously studied for algorithms different from MF. Large-scale recommender systems use techniques like min-hash clustering of users, probabilistic latent semantic indexing, and co-visitation counts to achieve fair scalability [30]. A more recent method based on multidimensional scaling embeds both the users and items in a common Euclidean space [31], reducing the retrieval task to the problem of k-nearest-neighbor search in Euclidean space. The plethora of algorithms for nearest-neighbor search can then be used for efficient RoR. While these methods show significant improvements in retrieval times, they deviate from the more accurate MF framework. The algorithms described in this Chapter were, at the time of publication, the first efficient retrieval algorithms in the MF framework. We address the problem of RoR in a MF framework and present two algorithms (one exact and one approximate) to solve (4.1) for multiple queries in a

sub-linear time complexity.

The main contributions on this chapter are:

- A simple branch-and-bound algorithm on a tree index with a novel bound to solve the exact problem.

- An approximate scheme that pre-computes solutions for certain representative queries and uses these solutions for the new queries.

- A theoretical error bound which determines the quality of the approximate results and used to control the approximation by adaptively rejecting overly (theoretically) inaccurate solutions.

Chapter 4 is based on our publication from [5].

4.1 Problem Definition

In MF models, the predicted rating of a user for an item boils down to a dot-product between two vectors representing the user and the item. Constructing the entire $\#USERS \times \#ITEMS$ preference matrix (or even just for the top-rated items for every user) requires heavy computational (and space) resources. For example, the Yahoo! Music dataset (described in Section3.2) has 1,000,990 users and 624,961 music items. Generating the optimal recommendations in this dataset requires over 6×10^{12} dot-products using a naive algorithm. A 50-dimensional model required 134 hours (over 5 days) to find optimal recommendations for all the users[1]. In terms of storage, saving the whole preference matrix requires over 5TB of disk-space. Moreover, this dataset of 10^6 users is just a small sample of the actual Yahoo! Music dataset and the problem worsens with larger numbers.

RoR in a trained MF model involves finding the set of K items for a user u with maximum predicted ratings. Equation 2.2 in Section 2.1 implies that for a given user u, ordering the items is independent of μ and the user's bias b_u. Thus, we can ignore these parameters without affecting the preferred ordering of the items and obtain an effective

[1] Using an Intel Xeon (E7320) CPU running at 2.13GHz.

rating:

$$\tilde{r}_{ui} = b_i + \mathbf{x}_u^\top \mathbf{y}_i. \tag{4.2}$$

It is important to note that this is only applicable during the RoR phase (after the training). Ignoring these components during the training will inevitably result in poor accuracy.

By appending the item bias to the item vector, the effective rating (equation 4.2) reduces to a simple dot-product:

$$\tilde{r}_{ui} = \tilde{\mathbf{x}}_u^\top \tilde{\mathbf{y}}_i = \|\tilde{\mathbf{x}}_u\| \ \|\tilde{\mathbf{y}}_i\| \cos\left(\theta_{\tilde{\mathbf{x}}_u, \tilde{\mathbf{y}}_i}\right), \tag{4.3}$$

where $\tilde{x}_u, \tilde{y}_i \in \mathbb{R}^{d+1}$ defined as $\tilde{x}_u = [x_u^\top \ 1]^\top$, $\tilde{y}_i = [t_i^\top \ b_i]^\top$, and $\theta_{\tilde{\mathbf{x}}_u, \tilde{\mathbf{y}}_i}$ is the angle between $\tilde{\mathbf{x}}_u$ and $\tilde{\mathbf{y}}_i$ at the origin.

Equation 4.3 implies that for a given user vector $\tilde{x}_u$, the items ordering is independent of the norm $\|\tilde{x}_u\|$ and only depends on the user vector through the angle $\theta_{\tilde{x}_u, \tilde{y}_i}$. Hence, without loss of generality, we normalize the user vectors to a unit vector $\bar{x}_u = \tilde{x}_u / \|\tilde{x}_u\|$ to further simplify equation 4.3 to:

$$\bar{r}_{ui} = \bar{\mathbf{x}}_u^\top \tilde{\mathbf{y}}_i = \|\tilde{\mathbf{y}}_i\| \cos\left(\theta_{\bar{p}_u, \tilde{q}_i}\right). \tag{4.4}$$

Note that while we normalize the concatenated user vectors, the concatenated item vectors can not be normalized without loss of accuracy.

For readability purposes, in the remaining of this chapter we will denote the concatenated user vector $\bar{\mathbf{x}}$ and item vector $\tilde{\mathbf{y}}$ as $\mathbf{x}_u$ and $\mathbf{y}_i$ respectively, and the effective rating for the task of retrieval as r_{ui}. We will keep to this notation from here onwards through the rest of Chapter 4. RoR reduces to the following task: Given a user query $\mathbf{x}_u$, we want to find an item $\mathbf{y}_i \in S$ such that:

$$\mathbf{x}_u^\top \mathbf{y}_i = \max_{\mathbf{y} \in S} \mathbf{x}_u^\top \mathbf{y} \tag{4.5}$$

Hence the RoR task is equivalent to the problem of finding the best-match for a query

$\mathbf{x}_u$ in a set of points with respect to the dot-product.

As shown in Chapter 3, it is possible to incorporate additional information such as taxonomy and temporal dynamics by adding auxiliary vectors to the item and user vectors. These additional components are usually additive and can be collapsed to the form of (4.4) using a very similar process. Without loss of generality we will therefore only discuss the basic model (equation 4.5). The extension to more complex models is usually trivial and in fact, we do use the complex model presented in Chapter 3 in the evaluation stage of this chapter.

4.2 Nearest Neighbor Search Algorithms

Efficiently finding the best match using the dot-product (Equation 4.5) appears to be very similar to much existing work in the literature. Finding the best match with respect to the Euclidean (or more generally l_p) distance is the widely studied problem of nearest-neighbor search in metric spaces [32]. The nearest-neighbor search problem (in a metric space) can be solved approximately with the popular *Locality-sensitive hashing* (LSH) method [33]. LSH can be applied to other forms of similarity functions (as opposed to the distance as a dissimilarity function) like the cosine similarity [34]. Next, we investigate the main differences between the problem stated in equation 4.5 and these previous problems.

4.2.1 Nearest-neighbor Search in a Metric Space

The problem of finding the nearest-neighbor in this setting is to find a point $\mathbf{y}_i \in S$ for a query $\mathbf{x}_u$ such that:

$$\begin{aligned} \mathbf{y}_i &= \arg\min_{\mathbf{y}\in S} \|\mathbf{x}_u - \mathbf{y}\|^2 = \arg\max_{\mathbf{y}\in S} \left(\mathbf{x}_u^\top \mathbf{y} - \|\mathbf{y}\|^2/2\right) \\ &\neq \arg\max_{\mathbf{y}\in S} \mathbf{x}_u^\top \mathbf{y} \quad (\text{unless } \|\mathbf{y}\|^2 = const \ \ \forall \mathbf{y} \in S). \end{aligned}$$

Hence, if all the points in S are normalized to the same length, then the problem of finding the best match with respect to the dot-product is equivalent to the problem of nearest-neighbor search in any metric space. However, without this restriction, the two

problems can yield very different answers.

4.2.2 Cosine similarity

Retrieving the best match with respect to the cosine similarity requires finding a point $\mathbf{y}_i \in S$ for a query $\mathbf{x}_u$ such that:

$$\begin{aligned} \mathbf{y}_i &= \arg\max_{\mathbf{y}\in S} \frac{\mathbf{x}_u^\top \mathbf{y}}{\|\mathbf{x}_u\|\,\|\mathbf{y}\|} = \arg\max_{\mathbf{y}\in S} \frac{\mathbf{x}_u^\top \mathbf{y}}{\|\mathbf{y}\|} \\ &\neq \arg\max_{\mathbf{y}\in S} \mathbf{x}_u^\top \mathbf{y} \quad (\text{unless } \|\mathbf{y}\| = const \;\; \forall \mathbf{y} \in S). \end{aligned}$$

As in the previous case, the best match with respect to the cosine similarity is identical the best match with respect to the dot-product only if all the points in the set S are normalized to have the same length (norm). Under general conditions, the best matches with these two similarity functions can be very different.

4.2.3 Locality Sensitive Hashing (LSH)

LSH involves constructing hashing functions h which satisfy the following – for any pair of points $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$:

$$\Pr[h(\mathbf{x}) = h(\mathbf{y})] = \text{sim}(\mathbf{x}, \mathbf{y}), \tag{4.6}$$

where $\text{sim}(\mathbf{x}, \mathbf{y}) \in [0, 1]$ is the similarity function of interest. For our situation, we can scale our dataset such that $\forall\ \mathbf{y} \in S, \|\mathbf{y}\| \leq 1$. Note that this normalization is different than the normalization mentioned before. Here the lengths of all the vectors are normalized to be less than equal to one, but not equal to each other. Let us also assume that the data is in the first quadrant (such as in non-negative matrix factorization models [35]). In that case, $\text{sim}(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\top \mathbf{y} \in [0, 1]$ is our similarity function of interest.

For any similarity function to admit a locality sensitive hash function family (as defined in equation 4.6), the distance function $\mathbf{d}(\mathbf{x}, \mathbf{y}) = 1 - \text{sim}(\mathbf{x}, \mathbf{y})$ must satisfy the triangle inequality (Lemma 1 in [34]). However, the distance function $\mathbf{d}(\mathbf{x}, \mathbf{y}) = 1 - \mathbf{x}^\top \mathbf{y}$ does not satisfy the triangle inequality. Hence, LSH cannot be applied to the dot-product similarity function even under restrictive assumptions where we assume that all the data

lies in the first quadrant.

4.2.4 Maximum Dot-products

Unlike the distance functions in metric space, dot-products do not induce any form of triangle inequality (even under some assumptions as mentioned above). This lack of any induced triangle inequality causes the similarity function to have no admissible family of locality sensitive hashing functions. Any modification to the similarity function to conform to widely used similarity functions (like Euclidean distance or Cosine-similarity) will create inaccurate results. Moreover, dot-products lack the basic property of *coincidence* – the self similarity is highest. For example, the Euclidean distance of a point to itself is 0; the cosine-similarity of a point to itself is 1 (highest). But, the dot-product of a point $\mathbf{x}$ to itself is $\|\mathbf{x}\|^2$, which may be high or low depending on the value of $\|\mathbf{x}\|$. There can possibly be many other points $\mathbf{y}_i$ $(i = 1, 2, \ldots)$ in the set such that $\mathbf{x}^\top \mathbf{y}_i > \|\mathbf{x}\|^2$. Therefore without any assumptions, the problem of obtaining the best match with respect to the dot-product is inherently harder than the previously addressed similar problems.

4.3 Fast Exact Retrieval Using Metric Trees

In this section, we describe metric trees and develop a novel bound to use with a simple branch-and-bound algorithm to provide the first method to efficiently obtain the exact best-matches with respect to the dot-products.

4.3.1 Metric Trees

Metric trees [36] are binary space-partitioning trees that are widely used for the task of indexing datasets in Euclidean spaces. The space is partitioned into overlapping hyperspheres (balls) containing the points (figure 4.1). We use a simple metric tree construction heuristic that tries to approximately pick a pair of pivot points farthest apart from each other [37] and splits the data by assigning points to their closest pivot. The tree T is built hierarchically and each node in the tree is defined by the mean of the data in that node (T.center) and the radius of the ball around the mean enclosing the points in the

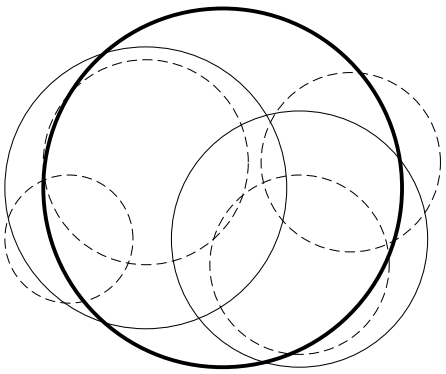


Figure 4.1: Metric-trees – note that while all the points in a child node lie also inside the parent ball, the child ball itself does not necessarily lie within the parent ball.

node (T.radius). The tree has leaves of size at most N_0. The splitting and the recursive tree construction algorithms are presented in Algorithms 3 & 4.

The tree is space efficient since every node only stores the indexes of the item vectors instead of the item vectors themselves. Hence the matrix for the items is never duplicated. Another implementation optimization is that the vectors in the items' matrix are sorted in place (during the tree construction) such that all the items in the same node are arranged serially in the matrix. This avoids random memory access while accessing all the items in the same leaf node.

4.3.2 Branch-and-bound Algorithm

Metric trees are used for efficient nearest neighbor search and are fairly scalable in moderately high dimensions [37, 38]. The search employs a depth-first branch-and-bound algorithm. A nearest-neighbor query is answered by traversing the tree in a depth-first manner– going down the node closer to the query first and bounding the minimum possible distance to items in other branches with the triangle-inequality. If this branch is farther away than the current neighbor candidate, the branch is removed from computation.

Since the triangle inequality does not hold for the dot-product, we present a novel analytical upper bound for the maximum possible dot-product of a user vectors with points (in this case, items) in a ball. We then employ a similar branch-and-bound algorithm

Algorithm 3 MakeMetricTreeSplit(Data S)

Pick a random point $\mathbf{v} \in S$
$\mathbf{c} \leftarrow \arg\max_{\mathbf{y}_i \in S} \|\mathbf{v} - \mathbf{y}_i\|$
$\mathbf{d} \leftarrow \arg\max_{\mathbf{y}_i \in S} \|\mathbf{c} - \mathbf{y}_i\|$
$\mathbf{w} \leftarrow (\mathbf{d} - \mathbf{c})$
$b \leftarrow -\frac{1}{2}\left(\|\mathbf{c}\|^2 - \|\mathbf{d}\|^2\right)$
return $(\mathbf{w}, b)$

Algorithm 4 MakeMetricTree(Set of items S)

Input – Set S
Output – Tree Q
$Q.S \leftarrow S$
Q.center $\leftarrow$ mean(S)
Q.radius $\leftarrow \max_{\mathbf{y}_i \in S} \|T.\text{center} - \mathbf{y}_i\|$
if $|S| \leq N_0$ **then**
  // Leaf node
  return Q
else
  // else split the set
  $(\mathbf{w}, b) \leftarrow$ MakeMetricTreeSplit(S)
  $S_l \leftarrow \{\mathbf{y}_i \in S : \mathbf{w}^\top \mathbf{y}_i + b \leq 0\}$
  $S_r \leftarrow S \setminus S_l$
  Q.left $\leftarrow$ MakeMetricTree(S_l)
  Q.right $\leftarrow$ MakeMetricTree(S_r)
  return Q
end if

Figure 4.2: Metric-tree Construction: The object $Q.S$ denotes the set of items in the node Q, Q.center denotes the Euclidean mean of the items in the node Q and Q.radius denotes the minimum radius of the ball centered around Q.center enclosing all the items in the node Q.

for the purpose of searching for the K-highest dot-products (as opposed to the minimum pairwise distance in K-nearest-neighbor search).

Bounding Maximal Inner-product

The following explanation is illustrated in Figure 4.3. Let $B^r_{\mathbf{y}_c}$ be the ball of items centered around $\mathbf{y}_c$ with radius r. Suppose that $\mathbf{y}^*$ is the best possible recommendation in the ball $B^r_{\mathbf{y}_c}$ for the user represented by the vector $\mathbf{x}_u$, and r^* be the Euclidean distance between the ball center $\mathbf{y}_c$ and the best possible recommendation $\mathbf{y}^*$ (by definition, $r^* \leq r$). Let

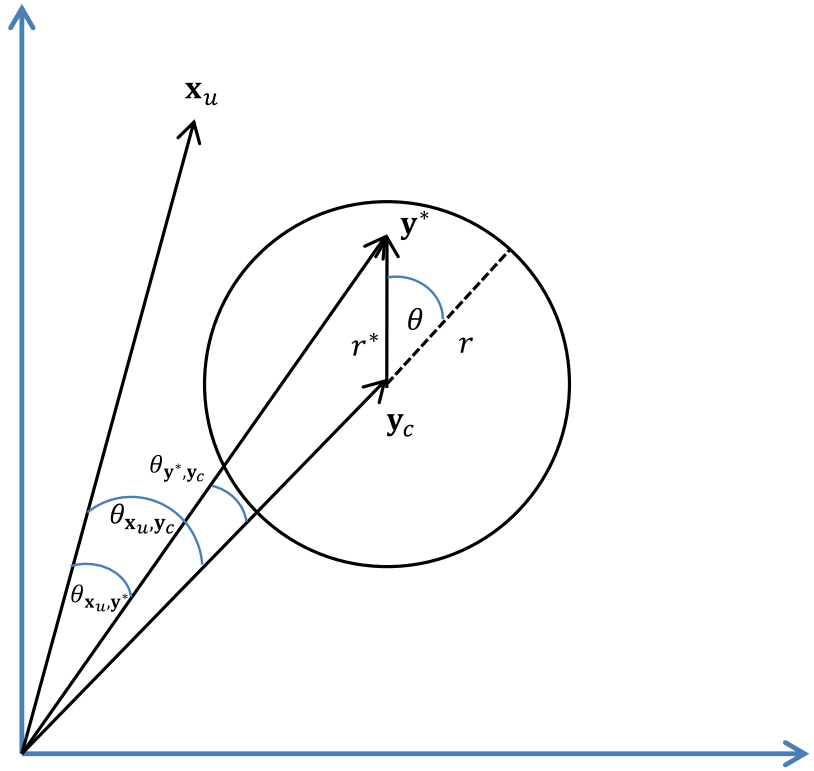


Figure 4.3: Bounding with a ball

θ be the angle between the vector $\vec{\mathbf{y}_c}$ and the vector $\mathbf{y}_c\vec{\mathbf{y}}^*$, $\theta_{\mathbf{x}_u,y_c}$ and $\theta_{\mathbf{y}^*,\mathbf{y}_c}$ be the angles between the vector $\vec{\mathbf{y}_c}$ and vectors $\vec{\mathbf{x}_u}$ and $\vec{\mathbf{y}^*}$ respectively. The distance of $\mathbf{y}^*$ from $\mathbf{y}_c$ is $r^* \sin\theta$ and the length of the projection of $\mathbf{y}^*$ onto $\mathbf{y}_c$ is $\|\mathbf{y}_c\| + r^* \cos\theta$. Therefore we have:

$$\|\mathbf{y}^*\| = \sqrt{(\|\mathbf{y}_c\| + r^* \cos\theta)^2 + (r^* \sin\theta)^2}, \tag{4.7}$$

$$\cos\theta_{\mathbf{y}^*,\mathbf{y}_c} = \frac{\|\mathbf{y}_c\| + r^* \cos\theta}{\|\mathbf{y}^*\|}, \sin\theta_{\mathbf{y}^*,\mathbf{y}_c} = \frac{r^* \sin\theta}{\|\mathbf{y}^*\|}. \tag{4.8}$$

Let $\theta_{\mathbf{x}_u,\mathbf{y}^*}$ be the angle between the vectors $\vec{\mathbf{x}_u}$ and $\vec{\mathbf{y}^*}$. This gives the following inequality regarding the angle between the user and the best possible recommendation (we assume that the angles lie in the range of $[-\pi, +\pi]$ instead of the usual $[0, 2\pi]$) :

$$|\theta_{\mathbf{x}_u,\mathbf{y}^*}| \geq |\theta_{\mathbf{x}_u,\mathbf{y}_c} - \theta_{\mathbf{y}^*,\mathbf{y}_c}|, \tag{4.9}$$

which implies

$$\cos\theta_{\mathbf{x}_u,\mathbf{y}^*} \leq \cos(\theta_{\mathbf{x}_u,\mathbf{y}_c} - \theta_{\mathbf{y}^*,\mathbf{y}_c}), \tag{4.10}$$

since $\cos(\cdot)$ is monotonically decreasing in the range $[0, \pi]$. Using this equality we obtain the following bound for the highest possible affinity between the user and any item within that ball:

$$\begin{aligned}
\max_{\mathbf{y}_i \in B^r_{\mathbf{q}_c}} \mathbf{x}_u^\top \mathbf{y}_i &= \mathbf{x}_u^\top \mathbf{y}^* (\text{by assumption}) \\
&= \|\mathbf{x}_u\| \ \|\mathbf{y}^*\| \cos\theta_{\mathbf{x}_u,\mathbf{y}^*} \\
&\leq \|\mathbf{x}_u\| \ \|\mathbf{y}^*\| \cos(\theta_{\mathbf{x}_u,\mathbf{y}_c} - \theta_{\mathbf{y}^*,\mathbf{y}_c}),
\end{aligned}$$

where the last inequality follows from equation 4.10. Substituting equations 4.7 & 4.8 in the above inequality, we have

$$\begin{aligned}
\max_{\mathbf{y}_i \in B^r_{\mathbf{q}_c}} \mathbf{x}_u^\top \mathbf{y}_i &\leq \|\mathbf{x}_u\| \left(\cos\theta_{\mathbf{x}_u,\mathbf{y}_c}(\|\mathbf{y}_c\| + r^* \cos\theta) + \sin\theta_{\mathbf{x}_u,\mathbf{y}_c}(r^* \sin\theta)\right) \\
&\leq ||\mathbf{x}_u|| \max_\theta \left(\cos\theta_{\mathbf{x}_u,\mathbf{y}_c}(\|\mathbf{y}_c\| + r^* \cos\theta) + \sin\theta_{\mathbf{x}_u,\mathbf{y}_c}(r^* \sin\theta)\right) \\
&= ||\mathbf{x}_u|| \left(\cos\theta_{\mathbf{x}_u,\mathbf{y}_c}(\|\mathbf{y}_c\| + r^* \cos\theta_{\mathbf{x}_u,\mathbf{y}_c}) + \sin\theta_{\mathbf{x}_u,\mathbf{y}_c}(r^* \sin\theta_{\mathbf{x}_u,\mathbf{y}_c})\right) \\
&\leq ||\mathbf{x}_u|| \left(\cos\theta_{\mathbf{x}_u,\mathbf{y}_c}(\|\mathbf{y}_c\| + r \cos\theta_{\mathbf{x}_u,\mathbf{y}_c}) + \sin\theta_{\mathbf{x}_u,\mathbf{y}_c}(r \sin\theta_{\mathbf{x}_u,\mathbf{y}_c})\right) \\
&\quad (\text{since } r^* \leq r).
\end{aligned}$$

The second inequality comes from the definition of maximum, and the next equality comes from maximizing over θ giving us the optimal value for $\theta = \theta_{\mathbf{x}_u,\mathbf{y}_c}$. Simplifying the final inequality gives us the following upper bound:

$$\max_{\mathbf{y}_i \in B^r_{\mathbf{y}_c}} \mathbf{x}_u^\top \mathbf{y}_i \leq \mathbf{x}_u^\top \mathbf{y}_c + r \|\mathbf{x}_u\| . \tag{4.11}$$

Algorithm 5 SearchMetricTree(User $\mathbf{x}_u$, Item Tree Node Q)

if $\mathbf{x}_u$.ub $< \mathbf{x}_u^\top Q$.center $+ Q$.radius $\cdot \|\mathbf{x}_u\|$ **then**
  // This node has potential
  if *isLeaf*(Q) **then**
    for each $\mathbf{y}_i \in Q.S$ **do**
      if $\mathbf{x}_u^\top \mathbf{y}_i > \mathbf{x}_u$.ub **then**
        $\mathbf{y}' \leftarrow \arg \min_{\mathbf{y} \in \mathbf{x}_u.\text{candidates}} \mathbf{x}_u^\top \mathbf{y}$
        $\mathbf{x}_u$.candidates $\leftarrow \{\mathbf{x}_u.\text{candidates} \setminus \{\mathbf{y}'\}\} \cup \{\mathbf{y}_i\}$
        $\mathbf{x}_u$.ub $\leftarrow \min_{\mathbf{y} \in \mathbf{x}_u.\text{candidates}} \mathbf{x}_u^\top \mathbf{y}$
      end if
    end for
  else
    // best depth first traversal
    $I_l \leftarrow \mathbf{x}_u^\top Q$.left.center; $I_r \leftarrow \mathbf{x}_u^\top Q$.right.center;
    if $I_l \leq I_r$ **then**
      SearchMetricTree($\mathbf{x}_u, Q$.right);
      SearchMetricTree($\mathbf{x}_u, Q$.left);
    else
      SearchMetricTree($\mathbf{x}_u, Q$.left);
      SearchMetricTree($\mathbf{x}_u, Q$.right);
    end if
  end if
end if
// Else the node is pruned from computation
return;

Algorithm 6 FindExactRecommendations(User $\mathbf{x}_u$, Item Tree Node Q)

$\mathbf{x}_u$.ub $\leftarrow 0$;
$\mathbf{x}_u$.candidates $\leftarrow \emptyset$;
SearchMetricTree($\mathbf{x}_u$, Q);
return $\mathbf{x}_u$.candidates;

Figure 4.4: Metric-tree Search: The object $\mathbf{x}_u$.candidates contains the set of current best K candidate items and x_u.ub denotes the lowest affinity between the user and its current best candidates.

Fast Retrieval Algorithm

Using this upper bound (4.11) for the maximum possible dot-product, we present the depth-first branch-and-bound algorithm to search for the K-highest dot-products in Algorithm 5. The algorithm begins at the root of the tree of items. At each subsequent step, the algorithm is at a tree node. Using the bound in equation 4.11, the algorithm checks if

the best possible item in this node is any better than the current best candidates for the user. If the check fails, this branch of the tree is not explored any more. Otherwise, the algorithm recursively traverses the tree, exploring the branch with the better potential candidates in a depth-first manner. If the node is a leaf, the algorithm just finds the best candidates within the leaf with the simple naive search. This algorithm ensures that the exact solution (i.e., the best candidates) is returned by the end of the algorithm.

Theoretical Runtime Bounds

We do not have any runtime guarantees for the algorithm presented in this section. However, we can conjecture possible runtime bounds. If the metric-tree constructed with Algorithm 4 has a depth of $O(\log |Q|)$ (where Q is the set of items), then the runtime bound for the construction of the tree is $O(d|Q| \log |Q|)$, where d is the dimensionality of the model (since it requires $O(d|Q|)$ operations at each level). During the tree-search algorithm (Alg. 6), let us assume that the user visits L leaves. If L is much smaller and in fact independent of the number of items $|Q|$, we can say that the runtime bound for the search process for a single user is $O(dL \log |Q|)$. However, if L depends on $|Q|$ as well, then the best possible runtime bound is $O(d|Q|)$.

Since algorithm 4 does not enforce that the splits be balanced, it is quite possible that the depth of the tree might end up being $O(|Q|)$, in which case, the worst case runtime for the search process is $O(d|Q|)$. However, in practice, the tree depths have been seen to be way less than $O(|Q|)$.

4.4 Fast Approximate Retrieval by Clustering Users

The efficiency of the exact algorithm can be limited, and some applications may require even faster retrieval while allowing for some suboptimal recommendations. To this end, we propose a scheme to cluster the users into *cones* of similar "taste". We then pre-compute the recommendations for the cone centers (representative user tastes), and use these recommendations as approximate recommendations for incoming users.

Equation 4.4 specifies that the user preferences depend on the angle (direction) of the

corresponding user vector but not on its norm (length). The smoothness of the cosine function implies that two users with vectors in similar directions will have very similar preferences. Hence, we partition the space into cones that aggregate users with similar taste. Let C be a set of cone centers where each $\mathbf{c} \in C$ is a unit vector. The direction of $\mathbf{c}$ is the taste of the cone which can be used to pre-compute recommendations for the users in that cone.

For a new user query $\mathbf{x}_u$, its best cone center $\mathbf{c}^*$ is:

$$\mathbf{c}^* \leftarrow \arg\max_{\mathbf{c} \in C} \mathbf{x}_u^\top \mathbf{c} \tag{4.12}$$

After finding $\mathbf{c}^*$ we retrieve the pre-computed recommendations of $\mathbf{c}^*$ as the approximated recommendations for $\mathbf{x}_u$. Figure 4.5 depicts a user's vector $\mathbf{x}_u$ and its best cone's vector $\mathbf{c}^*$. If Δ, the angle between $\mathbf{x}_u$ and $\mathbf{c}^*$, is small enough, then the approximated recommendations (based on $\mathbf{c}$) will be similar or even identical to the optimal recommendations. The speedup is achieved since the number of cones is much smaller than the number of users or items, thus finding $\mathbf{c}^*$ is significantly easier than computing the exact recommendations for each user .

The approximation is controlled by using an upper bound on the relative approximation error in terms of Δ (this is presented in section 4.4.1). This bound evaluates the quality of the approximation. By defining a threshold T_r on the maximum acceptable error, we adaptively accept the pre-calculated approximate results when the error bound is below the threshold, or compute the exact results otherwise. The details of the approximate RoR algorithm are given in figure 4.6. The threshold T_r introduces a tradeoff between speedup and accuracy where maximal speedup is achieved when $T_r = \infty$.

Choosing cones There are different schemes that can be used to choose a set of user-cones. In general, one would like to choose a set of user clusters (cones) which appropriately fits the distribution of possible queries. This can be efficiently achieved by spherical clustering of the user vectors. Spherical clustering defines groups of users with similar preferences or taste. Unit vectors in the direction of the clusters' centers define the cone

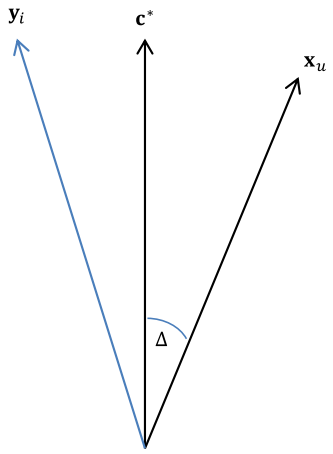


Figure 4.5: $\mathbf{x}_u$ is the user's vector, $\mathbf{c}^*$ is a vector in the direction of the cone's center, $\mathbf{y}_i$ is the item's vector and Δ is the angle between $\mathbf{x}_u^*$ and $\mathbf{c}$.

centers. We chose spherical clustering because of its computational efficiency. Furthermore, the clustering already assigns the user vectors into cones and there is no need to search for the best matching cone for the existing users.

Note that clustering assumes the presence of groups of users common tastes. This is a very natural assumption in every collaborative filtering algorithm.A requirement for a good clustering is that $\Delta < \frac{\pi}{2}$. Otherwise the dot-product between the user and an item in the direction of the cone's center can be negative, which implies that the user does not like the items that fit the cone's vector.

4.4.1 Approximation Error Bound

In this subsection we present a theoretical error bound on the relative approximation error for any user. This is used by the adaptive algorithm to control the approximation error.

The vector $\mathbf{y}_i$ in figure 4.5 depicts an item's vector that was chosen as an optimal recommendation based on the cluster's center $\mathbf{c}^*$. Intuitively, as Δ decreases, the approx-

Algorithm 7 PrepareCones(User vectors X, Item vectors S)

Input – Set X, S
Output – Cone centers C, Tree Q
C = ChooseCones(X);
Q = MakeMetricTree(S);
for all $\mathbf{c} \in C$ **do**
  $\mathbf{c}$.candidates = FindExactRecommendations($\mathbf{c}, Q$);
end for

Algorithm 8 FindApproxRecommendations(User $\mathbf{x}_u$, Cone centers C, Threshold T_r, Item Tree Q)

$\mathbf{c}^* \leftarrow \arg\max_{\mathbf{c} \in C} \mathbf{x}_u^\top \mathbf{c}$;
$ErrorBound$ = ComputeErrorBound($\mathbf{c}^*, \mathbf{x}_u$);
if $ErrorBound \leq T_r$ **then**
  return $\mathbf{c}^*$.candidates;
else
  return FindExactRecommendations($\mathbf{x}_u, Q$);
end if

Figure 4.6: Approximate RoR: The subroutine *ChooseCones* chooses a set of cones that fits the set of user vectors X in the dataset. Then, the optimal recommendations are computed for each cone's center using the metric tree of section 4.3.
In *FindApproxRecommendations*, the subroutine *ComputeErrorBound* computes the error bound according to equation 4.15. The approximated recommendations are used for every query with an error bound below the error threshold T_r, otherwise exact recommendations are computed.

imation error should decrease as well. We define the approximation error $err = exp - real$ as the difference between the expected rating based on the cluster $exp = \mathbf{y}_i^\top \mathbf{c}^*$ and the real dot-product with the user's trait vector $real = \mathbf{y}_i^\top \mathbf{x}_u$. The relative error is then:

$$\frac{err}{exp} = \frac{exp - real}{exp} = 1 - \frac{real}{exp} \tag{4.13}$$

We assume here that for every cone $exp > 0$; otherwise it means that there are no fitting recommendation for that cone, which is very unlikely and we never encountered this[2].

[2] Even if $exp < 0$ it is still possible to bound the error by following a very similar process to the one shown here.

Since we want the worst-case bound, we ignore the case where $real > exp$ since this situation means that the affinity between $\mathbf{x}_u$ and $\mathbf{y}_i$ is better than expected. Hence, assuming that $real < exp$, we have the following:

$$\frac{real}{exp} = \frac{\cos\left(\theta_{\mathbf{x}_u,\mathbf{y}_i}\right)}{\cos\left(\theta_{\mathbf{c}^*,\mathbf{y}_i}\right)} \geq \frac{\cos\left(\theta_{\mathbf{c}^*,\mathbf{y}_i} + \Delta\right)}{\cos\left(\theta_{\mathbf{c}^*,\mathbf{y}_i}\right)} \tag{4.14}$$

The inequality follows from the fact that $\theta_{\mathbf{c}^*,\mathbf{y}_i} \leq \frac{\pi}{2}$ (because $exp \geq 0$) and $\Delta \leq \frac{\pi}{2}$ (a requirement of the clustering). Since $\cos(\cdot)$ is monotonically decreasing in the range $[0, \pi]$, we get $\cos\left(\theta_{\mathbf{x}_u,\mathbf{y}_i}\right) \geq \cos\left(\theta_{\mathbf{c}^*,\mathbf{y}_i} + \Delta\right)$. Substituting (4.14) into (4.13) we get the following upper bound on the relative error:

$$\frac{err}{exp} \leq 1 - \frac{\cos\left(\theta_{\mathbf{c}^*,\mathbf{y}_i} + \Delta\right)}{\cos\left(\theta_{\mathbf{c}^*,\mathbf{y}_i}\right)} \tag{4.15}$$

Note that this bound is a tight bound. Namely, when $\Delta \to 0$ we get $err \to 0$.

4.5 Experiments and Evaluations

We present the results of the exact RoR algorithm (Section 4.3) and the approximate RoR algorithm (Section 4.4) demonstrating its efficiency-error tradeoff. Finally, as a thought experiment, we also present the inaccuracies introduced by using existing best-match algorithms (nearest-neighbor search in Euclidean space and best-match with respect to cosine similarity) for the task of RoR.

4.5.1 Datasets

We used the following publicly available datasets:

1. MovieLens – Consisting of 1,000,206 ratings of 3,952 movies by 6,040 users. Ratings are integers in the range 1-5, and the dataset is 95.81% sparse.

2. Netflix [19] – Consisting of 100,480,507 ratings of 17,770 movies by 480,189 users. The ratings in Netflix are on a scale of 1-5, and the dataset is 98.82% sparse.

3. Yahoo! Music [2] – This dataset is the dataset from Chapter 3. It is the largest of the three consisting of 252,800,275 ratings of 624,961 music items by 1,000,990 users. The ratings are on a scale of 0-100 and the dataset is 99.96% sparse.

Currently the Yahoo! Music dataset of Chapter 3 is the largest publicly available collaborative filtering dataset. Both our algorithms perform best on this dataset. In fact, in most of our evaluations the results seems to improve with the size of the dataset. This is expected as overhead times become negligible when the number of queries (users) increase. All the above datasets were in fact sampled from real datasets which were possibly much larger. It is therefore likely that the results presented in this paper will further improve when the proposed algorithms are implemented in real world systems.

For the MovieLens and Netflix datasets, we built and trained a basic MF model ((2.1) in Section 2.1) using stochastic gradient descent minimization of the mean squared error. For the Yahoo! Music dataset, we used the model presented in Chapter 3 that incorporates music taxonomy and temporal effects. All models were trained with 50-dimensional latent vectors. The root mean squared errors of these three models were 0.839 in MovieLens, 0.899 in Netflix, and 22.592 in Yahoo! Music.

We quantify the improvement of an algorithm A over another (baseline) algorithm A_0 by the following term:

$$\text{Speedup}_{A_0}(A) = \frac{\text{Time taken by Algorithm } A_0}{\text{Time taken by Algorithm } A}. \tag{4.16}$$

Since there are no efficient search algorithms for maximum dot-products, our baseline is a naive algorithm that searches over all items to find the best recommendations for every user. We denote by T_{naive} the time taken by the naive algorithm. It is clear that ranking the items in a naive approach is $\Theta(\#USERS \times \#ITEMS \times d)$, where d is the dimensionality of the vectors (here $d = 50$).

As expected, the naive algorithm is extremely time consuming. For example, the baseline execution time for retrieving optimal recommendations for the Yahoo! Music

dataset is 135.1 hours[3] (over 5 days). The mean latency for a single user query is 0.482 seconds. Using our proposed combined method (figure 4.6), we achieve up to ×258.08 speedup, which is equivalent to just 31.4 minutes for the entire computation or an average single user latency of 1.87 milliseconds. It is important to note that while the overall computation time can also be reduced by means of parallelization, the latency for a single user might be harder to improve upon.

Implementation Details We used the *Cluto* clustering toolkit [39] for spherical clustering of the user vectors. We used 500, 1000, and 2000 clusters (cones) for the MovieLens, Netflix and Yahoo! Music respectively, because these values showed a good balance between performance and speedup. In Alg. 4, we used $N_0 = 2$ in all our experiments. In general, these parameters can be optimized using a cross-validation process.

4.5.2 Exact RoR

The time taken by the exact algorithm of Section 4.3 can be broken up into two parts as follows:

$$T_{exact} = T_{tree\ building} + T_{tree\ search}, \tag{4.17}$$

where $T_{tree\ building}$ is the time taken by Alg. 4 to build the tree on the set of item vectors, and $T_{tree\ search}$ is the time taken to find the best recommendations to all users using Alg. 5. The speedup is therefore:

$$\text{Speedup}_{naive}(exact) = \frac{T_{naive}}{T_{exact}}. \tag{4.18}$$

We present the speedups obtained for different numbers of top recommendations in Table 4.1. The results indicate that the exact algorithm can be up to ×7 faster than the naive algorithm. Another advantage of this method is its space efficiency – only the tree (which consists solely of pointers) has to be stored. The complete ($\#USERS \times \#ITEMS$) user-preference matrix does not have to be stored and the recommendations for a user can be obtained when required.

[3] Using an Intel Xeon (E7320) CPU running at 2.13GHz

Dataset	$K = 1$	$K = 5$	$K = 10$	$K = 50$
MovieLens	3.01	1.82	1.73	1.21
Netflix	2.87	2.39	1.95	1.31
Yahoo! Music	7.26	5.25	4.7	3.01

Table 4.1: Speedups of Alg. 5 over naive search for different number of top recommendations (K).

An important thing to note is that the tree-building task is extremely time efficient – for example, for the Yahoo! Music dataset, the time taken to build the metric-tree on the set of items (of size 624961×50) was less than 16 seconds (the time required to load the whole data into memory took more than 40 seconds!). The tree-building process is a one-time cost which is amortized by the more expensive tree-search process. Moreover, new items can be easily added to this metric-tree index[4]. Nevertheless, we include the tree-building times in our computation for completeness.

It is important to note that the search time increases with K (the number of top recommendations returned) and as a result the speedup decreases. This is because the worst candidate for the user (x_u.ub in Alg. 5) generally becomes smaller (worse) with increasing K (Line 8 in Alg. 5). Hence, the number of nodes that have potential (Line 1 in Alg. 5) increase and so does the number of leaves finally visited.

Some applications may require more than just the top 50 items. In that case, the tree-based exact search does not provide any significant improvement over the naive algorithm. Therefore, we present further improvements in computational performance in the next subsection with the proposed approximate algorithm.

4.5.3 Approximate RoR

The time taken by the approximate algorithm of Section 4.4 can be broken up into four parts as follows:

$$T_{approx} = T_{clustering} + T_{tree\ building} + T_{search\ cones} + T_{search\ queries},$$

[4] Efficient item insertion is inherent to tree data structures.

MovieLens

Threshold	0.25	0.5	0.75	∞
K=1	x2.49	x7.26	x9.02	x9.25
K=5	x1.6	x5.68	x7.66	x7.94
K=10	x1.52	x5.5	x7.49	x7.78
K=50	x0.89	x3.82	x7.73	x6.04

Netflix

Threshold	0.25	0.5	0.75	∞
K=1	x2.69	x5.65	x12.61	x17.29
K=5	x2.48	x5.3	x12.27	x17.26
K=10	x1.93	x4.29	x11.15	x17.18
K=50	x1.19	x2.81	x8.89	x16.97
K=500	x1.04	x2.5	x8.28	x16.89

Yahoo! Music

Threshold	0.25	0.5	0.75	∞
K=1	x10.49	x19.7	x150.87	x258.08
K=5	x7.67	x14.54	x128.46	x251.27
K=10	x6.88	x13.08	x120.77	x248.46
K=50	x4.45	x8.53	x91.78	x234.6
K=500	x1.49	x2.89	x39.1	x178.64

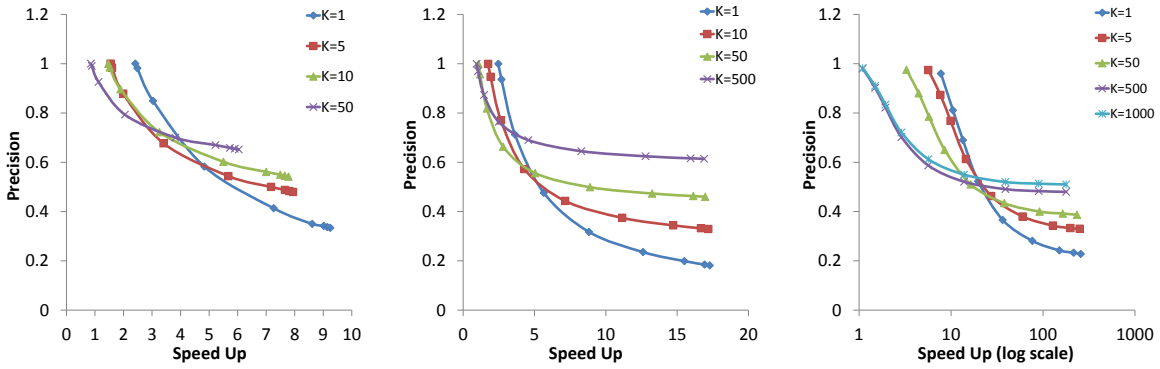
Table 4.2: Speedups of Alg. 8 over the naive algorithm for different values of K and the error threshold.

where $T_{clustering}$ is the time taken by the clustering algorithm, $T_{tree\ building}$ is the metric-tree construction time, $T_{search\ cones}$ is the search time for optimal recommendations for all the cones and, $T_{search\ queries}$ is the time taken to compute exact recommendations for queries that are above the threshold. The speedup of the approximate algorithm is:

$$\text{Speedup}_{naive}(approx) = \frac{T_{naive}}{T_{approx}}. \tag{4.19}$$

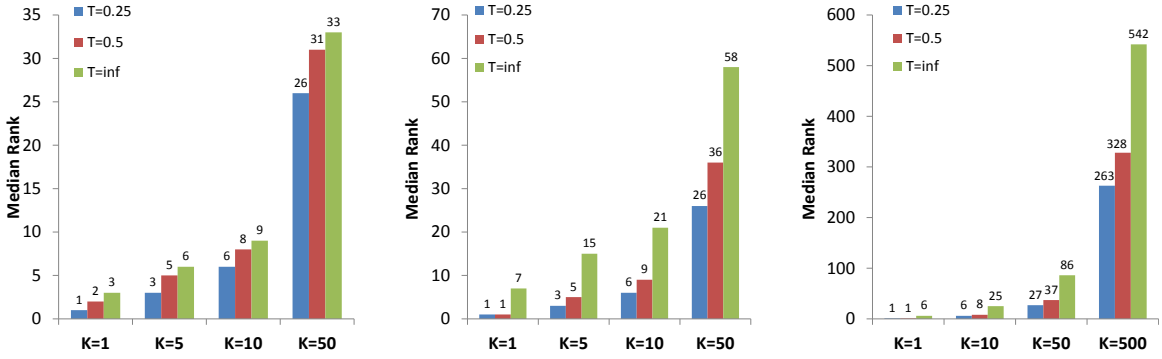
We define two terms to quantify the quality of the top K recommendations retrieved by the approximated method. The first quantity ($Precision$) denotes how similar the approximate recommendations are to the actual top K recommendations (which are retrieved by the naive approach):

$$Precision(K) \triangleq \text{mean}_u \left\{ \frac{|L_{rec}(u) \cap L_{opt}(u)|}{K} \right\}, \tag{4.20}$$



(a) MovieLens (b) Netflix (c) Yahoo! Music

Figure 4.7: Precision(K) of L_{rec} vs. speedup of the approximate algorithm. The error bound threshold defines a tradeoff between high precision to high speedup. We used higher values of K for datasets with more items.



(a) MovieLens (b) Netflix (c) Yahoo! Music

Figure 4.8: $MedianRank(K)$ of the adaptive algorithm for different values of the error bound threshold. Speedup values can be retrieved from table 4.2.

where $L_{rec}(u)$ and $L_{opt}(u)$ are the lists of the top K approximate and the top K optimal recommendations for the user u, respectively. Our evaluation metrics only care about the items at the top of the approximated and optimal lists ($L_{rec}(u)$ and $L_{opt}(u)$). In that case there is no real meaning to compute *Recall* because its natural definition would be identical to the *Precision*.

In addition, we define a secondary metric ($MedianRank$) which denotes the preference of the approximated recommendations with respect to the rest of the items:

$$MedianRank(K) \triangleq \text{median}\left\{\cup_u Rank(L_{rec}(u))\right\}, \tag{4.21}$$

where the function $Rank(L(u))$ returns a list of the optimal ranks for the items in $L(u)$

for user u (for example, $Rank(L_{opt}(u)) = \{1, 2, \ldots, K-1, K\}$).

A high value for $Precision$ implies that the approximate recommendations are very similar to the optimal recommendations, and a low value of $MedianRank$ implies that the approximate recommendations are highly preferred by the users. In many practical applications, it is very likely to have a low value for $Precision$ as well as for $MedianRank$. This implies that the items recommended by the approximate algorithm are generally different from the optimal items for the users, but the items recommended are still very highly preferred by the users.

The speedups of the approximate algorithm for different values of the error bound threshold are summarized in Table 4.2. The results indicate that the approximate RoR method can be up to $\times 258$ faster than the naive approach. The approximation quality for different levels of speedup is depicted in figures 4.7 & 4.8.

Figure 4.7 shows the tradeoff between precision and speedup achieved by using different values of the error bound threshold. When the threshold is high, the approximated result is less likely to be rejected. In this case, the precision is lower, speedup is higher, and performance is better for higher values of K. The latter is a result of the fact that precision in a finite set is easier to achieve as K is higher. When the threshold is low, the approximated result is more likely to be rejected. In this case, the precision is higher, speedup is lower, and performance is better when K is lower. The latter is a result of the fact that we are more likely to fall back to using the metric tree and the fact that the tree performs worse on higher values of K (as explained earlier).

Figure 4.8 presents the $MedianRank$ for different values of K and different values of the error bound threshold. Speedup values can be retrieved from table 4.2. We see that even when $Precison$ values are low (e.g., when $T_r = \infty$) the $MedianRank$ values are also relatively low, which indicate that the approximated recommendations are still highly preferred by the users.

Dataset	$K = 1$	$K = 5$	$K = 10$	$K = 50$	$K = 100$
MovieLens	0.4	0.54	0.59	0.72	0.77
Netflix	0.19	0.24	0.28	0.35	0.39
Y! Music	0.055	0.08	0.08	0.112	0.133

Table 4.3: Precision of the top (K) best matches with respect to the l_2 distance

4.5.4 Existing Best-Match Algorithms

In this subsection we perform an experiment to demonstrate that existing nearest-neighbor search algorithms (like LSH) cannot be applied directly to the task of RoR in the *existing MF framework* without introducing high levels of error. We find the top recommendations for a user with respect to the Euclidean (l_2) distance and with respect to the cosine similarity. The first returns the K items closest to the user (in terms of the l_2 distance), and the second returns the K items with the smallest angles between the user vector and the item vector (hence returning best matches with respect to the cosine similarity).

Dataset	$K = 1$	$K = 5$	$K = 10$	$K = 50$	$K = 100$
MovieLens	0.05	0.12	0.16	0.35	0.46
Netflix	0.14	0.24	0.31	0.48	0.56
Y! Music	0.004	0.01	0.014	0.033	0.047

Table 4.4: Precision of the top (K) best matches with respect to the cosine similarity

Tables 4.3 & 4.4 report the precision of the exact best-matches obtained with respect to Euclidean distance and cosine similarity respectively. As expected from our discussion in section 4.2, the precision is very low (especially on the larger Yahoo! Music dataset). Contrasting these numbers to the precision of the approximate solutions obtained from Alg. 8 (figure 4.7), we see that our approximate algorithm performs as accurately (if not better) with significant amount of speedup. For example, for the Yahoo! Music data set with $K = 50$, the best-matches with l_2 distance and cosine similarity have a precision of 0.112 and 0.033 respectively. In contrast, our proposed algorithm shows a speedup of about $\times 200$ while achieving a precision level of around 0.4 (figure 4.7(c)).

It is important to note that these returned recommendations in both cases (l_2 distance and cosine-similarity) are the *exact best-matches with respect to their corresponding sense*

of similarity. When the exact results are so inaccurate (in terms of recommendation quality), it is hard to expect good results once approximate techniques for these best-match problems like LSH are used.

V Item Based Recommendations

Most research on recommender systems is focused on modeling relations between users and items. New items are recommended to users based on their past purchases or activities. However, in reality user profiles are often not available (e.g., when a user is new or not logged-in) or irrelevant (e.g., when the current, short-term interest is unrelated to longer term inclinations). Therefore, many industrial systems are based on item-oriented recommendations, where item suggestions are solely based on their relatedness to a small set of currently considered items. Such recommendations aim at highlighting alternative items or items that are often bought together. A well known example of such a recommender is Amazon's shopping cart recommender. Amazon's system capitalizes on customers impulse buying patterns [40] – i.e., when a customer is currently considering a specific book, the system suggests to her more books that are often sold together. This real-world popular recommendation setup is characterized by inferring the recommendations based on the user's current interest rather than on her long term activity history. Often, these item-oriented systems employ signals like "users who liked this product also liked...", which are directly mined from usage logs.

While evidentially used in practice, we are not aware of published scientific works addressing collaborative-filtering item-oriented recommendations. This is not very surprising, as the problem seems rather simplistic compared to personalized recommendations, which indeed may use item-item relations internally. After all, one could argue that simply counting co-usage patterns among items can almost directly deliver the required similarity scores in the form of item-item conditional purchase probabilities, or other pairwise similarity metrics (e.g., cosine and Jaccard similarities, etc.). However, we wish to highlight two shortcomings of such straightforward retrieval techniques. First, point-wise counting of item-item co-occurrences requires an adequate support for both

items. Hence, similarities related to items subject to less user activity cannot be estimated reliably. Second, a naive retrieval of most similar items requires evaluating all possible items for each target item. Therefore, producing all related item pairs requires a quadratic time which is hard to scale as item inventories grow in size. The scalability issue could be alleviated by different pruning rules or by using items' hierarchy, yet in this Chapter we suggest a more systematic solution which accounts for both scalability challenges as well accuracy challenges in the tail.

The main contribution of our research with regard to item based recommendations is Euclidean Item Recommender (EIR) – a new algorithm for identifying related item-pairs, which addresses the two aforementioned shortcomings. Instead of determining item-item relations by the limited number of co-occurrences, we represent item-item conditional probabilities through latent factor vectors which are learned through a global learning algorithm utilizing the entire training data rather than just the pairwise information. We thus achieve a smooth estimation of similarities which facilitates more reliable similarities even for rarely co-purchased item pairs. Furthermore, the latent factor vectors are embedded in a Euclidean space instead of an inner product space. By switching from an inner product space to an Euclidean space we allow fast retrieval of most similar items using "traditional" nearest-neighbor algorithms without the need for special retrieval algorithms data structures such as in Chapter 4. We thus present an alternative approach to improve scalability at the retrieval phase - this time by changing the model itself to allow the use of more traditional data structures. Finally, another contribution of this chapter is the establishment of a modeling and experimentation scheme for item-oriented recommendation, which employs existing user preference datasets.

5.1 Related Work

Item-item neighborhood methods are well studied in the academic literature [40, 41]. We share with these works the need to derive pairwise relations between items. However, our research is item-oriented rather than user-oriented. Unlike previous work, we do not model the users in our dataset. We focus on modeling the item-item relations directly.

Some prior works [42, 43] used latent factors for representing item-item relations. This work also follows this path with several distinctions: First, our goal of item-oriented recommendation dictates a different cost function and training method. We present a probabilistic model for items co-occurrences that use latent item vectors to model the probabilities of any items pair to co-occur together. The use of a latent item vectors allows us to generalize the model also to rare combinations of items that were not seen in the training set. Second, we emphasize fast retrieval which is facilitated by embedding the factor vectors in a Euclidean space, unlike the inner-product space used by others. By utilizing an Euclidean similarity space we bypass the retrieval difficulties discussed in Chapter 4.

The use of Euclidean similarity spaces in recommender systems can be extended also to user-item algorithms. Therefore, this chapter also deals with the retrieval problem. Instead of suggesting new algorithms for fast retrieval in an inner product space (as in Chapter 4), here we introduce a new model that facilitates traditional retrieval algorithms in Euclidean spaces (e.g., KD-Trees or LSH). The use of Euclidean space instead of an inner product space was proposed also by Khoshneshin *et al.* [44]. There are two key differences between our work and theirs: First, as explained above, we deal with item-based recommendations. Second, while [44] used biases in their model, they ignore the biases at the retrieval phase and therefore they never actually demonstrated fast retrieval, unless biases are ignored. But biases are a dominant factor in recommendations, and ignoring them altogether would be a mistake. In Section 5.3 we explain how to perform fast retrieval even in the presence of item biases which are added on top of the Euclidean similarity.

5.2 Modeling Pairwise Item Relations

We represent item-item relations through their conditional probabilities. That is, given items i and j we will estimate $P(j|i)$, the conditional probability that a user consuming item i will consume item j as well. The conditional probabilities will be learned by embedding all items in a low-dimensional Euclidean space. An item k will be represented by

a d-dimensional vector $\mathbf{y}_k \in \mathbb{R}^d$, and a scalar bias b_k. The latent item vectors are designed to capture item similarities and the biases capture popularity patterns independently of other co-consumed items. To this end, we define the conditional probability $P(j|i)$ by the multinomial distribution

$$P(j|i) = \frac{\exp(-\|\mathbf{y}_i - \mathbf{y}_j\|^2 + b_j)}{\sum_k \exp(-\|\mathbf{y}_i - \mathbf{y}_k\|^2 + b_k)}. \tag{5.1}$$

Note that we assure $\sum_j P(j|i) = 1$. Finally, given a training set $\mathcal{D}$, containing item pairs of co-consumed (or co-liked) items, we seek to learn model parameters that maximize the log-likelihood of the training set:

$$\text{log-likelihood}\{\mathcal{D}\} \stackrel{\text{def}}{=} \sum_{(i,j)\in\mathcal{D}} \log P(j|i)\,. \tag{5.2}$$

5.2.1 Likelihood Maximization

Learning proceeds by stochastic gradient ascent. Given a training pair (i, j) we update each parameter θ (a latent vector component or a bias) by

$$\Delta\theta = \eta\frac{\partial P(j|i)}{\partial\theta} = \eta\left[\frac{\partial}{\partial\theta}\Big(-\|\mathbf{y}_i - \mathbf{y}_j\|^2 + b_j\Big) + \sum_k P(k|i)\frac{\partial}{\partial\theta}\Big(\|\mathbf{y}_i - \mathbf{y}_k\|^2 - b_k\Big)\right], \tag{5.3}$$

where η is the learning rate. However, such a training scheme would be too slow in practice as each update rule requires summing over all items. We thus resort to sampling the weighted sum in (5.3) based on the importance sampling idea proposed by Bengio and Senécal [45]; see also Aizenberg et al. [46] for a related usage of the technique.

With importance sampling we draw items according to a *proposal distribution*. In our case we assign each item a probability proportional to its empirical frequency in the training set (fraction of train pairs containing the item), and denote this proposal distribution by $P(i|\mathcal{D})$. Items are sampled with replacement from $P(i|\mathcal{D})$ into a list $\mathcal{L}$. Using $\mathcal{L}$, we approximate $P(k|i)$ for each $k \in \mathcal{L}$ with the weighting scheme

$$w(k|i) = \frac{\exp(-\|\mathbf{y}_i - \mathbf{y}_k\|^2 + b_k)/P(k|\mathcal{D})}{\sum_{l\in\mathcal{L}} \exp(-\|\mathbf{y}_i - \mathbf{y}_l\|^2 + b_l)/P(l|\mathcal{D})}. \tag{5.4}$$

Consequently, the approximated gradient ascent step given a training pair (i, j) will be

$$\Delta\theta = \eta \left[\frac{\partial}{\partial\theta}\Big(- \|\mathbf{y}_i - \mathbf{y}_j\|^2 + b_j \Big) + \sum_{k\in\mathcal{L}} w(k|i) \frac{\partial}{\partial\theta}\Big(\|\mathbf{y}_i - \mathbf{y}_k\|^2 - b_k \Big) \right]. \tag{5.5}$$

As mentioned in [45], it is desirable that the size of the set $\mathcal{L}$ grows as the training process proceeds because at later training phases more delicate parameter adjustments are needed. Hence, we employ a simple rule for controlling the sample size ($|\mathcal{L}|$) based on the fitness of the current estimate. Given a training pair (i, j), we keep sampling items into $\mathcal{L}$ until the following condition is satisfied:

$$\begin{gathered} \sum_{k\in\mathcal{L}} P(k|i) > \alpha \cdot P(j|i) \\ \Leftrightarrow \\ \sum_{k\in\mathcal{L}} \exp(-\|\mathbf{y}_i - \mathbf{y}_k\|^2 + b_k) > \alpha \cdot \exp(-\|\mathbf{y}_i - \mathbf{y}_j\|^2 + b_j) \end{gathered} \tag{5.6}$$

The adaptive sampling automatically lets the sample size grow when parameters are nearing final values and the correct paired item is getting a relatively high probability. In our implementation we used $\alpha = 3$. We impose a minimal size of 5 on the sample size. For efficiency we also limit the maximal sample size to 500.

5.3 Fast Retrieval

As explained in Chapter 4, retrieval time of recommendations is a key factor when designing real-world large-scale systems that need to address tens of thousands of queries per second. A major design goal of EIR is enabling fast pairing of items. In this setting, the task of efficiently retrieving recommendations requires finding items that the user is most likely to purchase given the item she is currently considering. Namely, we wish to find an item j that maximizes:

$$\max_{j\neq i} P(j|i) \quad \Leftrightarrow \quad \max_{j\neq i} -\|\mathbf{y}_i - \mathbf{y}_j\|^2 + b_j. \tag{5.7}$$

It is easy to see that by ignoring biases we are left only with the squared Euclidean distance between the two items vectors, and the retrieval is reduced to a simple nearest neighbor task. Nevertheless, biases are a key contributor to recommendations accuracy and should not be dismissed. We therefore propose a simple transformation to reduce the problem in (5.7) to that of a simple Euclidean search. For each item vector $\mathbf{y}_j$, we define a concatenated item vector $\hat{\mathbf{y}}_j$ as follows:

$$\hat{\mathbf{y}}_j = [\mathbf{y}_j^\top,\ \sqrt{M_b - b_j}]^\top \in \mathbb{R}^{d+1}, \tag{5.8}$$

where M_b is the maximum bias ($M_b = \max_j b_j$). We also define concatenated query vector as follows:

$$\bar{\mathbf{y}}_i = [\mathbf{y}_i^\top,\ 0]^\top. \tag{5.9}$$

It can be easily shown that (5.7) is equivalent to:

$$\max_{j \neq i} P(j|i) \quad \Leftrightarrow \quad \min_{j \neq i} \|\bar{\mathbf{y}}_i - \hat{\mathbf{y}}_j\|^2 = \min_{j \neq i} \|\mathbf{y}_i - \mathbf{y}_j\|^2 + M_b - b_j = \max_{j \neq i} -\|\mathbf{y}_i - \mathbf{y}_j\|^2 + b_j. \tag{5.10}$$

Therefore, this transformation facilitates a variety of Euclidean nearest neighbor algorithms e.g., Metric Trees, or Locality Sensitive Hashing (LSH).

5.4 Empirical Study

5.4.1 Dataset Construction

We evaluate our algorithm using four different datasets: The Netflix dataset [19], The Million Song Dataset (MSD) [47], Ziegler's dataset of books' reviews[1] [48], and the Yahoo! Music dataset from Chapter 3. Our work is focused on implicit ratings, however of the four aforementioned datasets only the MSD dataset is implicit. Therefore, we simulated imlicit data from the explicit datasets as follows: In Netflix, we first filtered only the ratings with a value ≥ 4. We then produced a dataset of "co-liked" movies – namely,

[1] `www.informatik.uni-freiburg.de/~cziegler/BX/`

	Items	Users	Training Examples	Test Examples
Netflix	17,749	478,488	53,414,617	941,614
MSD	384,526	1,019,296	45,078,691	2,037,890
YMusic	433,903	497,881	40,362,704	984,162
Books	340,536	103,723	1,068,842	66,306

Table 5.1: Datasets Statistics

movies that were liked by the same users. For the YMusic dataset, we repeated the same process for ratings ≥ 80. In the books dataset we simply used all the data-entries in order to create a dataset of co-consumed books that were read and reviewed by the same users (regardless of the review). Finally, in all datasets we generated item pairs as follows: For each user we created a random cyclic order of the items consumed by her. Then, the final dataset is comprised from all the pairs of consecutive items.

We split our datasets into train and test subsets by randomly choosing a subset of users and placing all their item pairs in the test-set. Table 5.1 summarize the final datasets statistics.

5.4.2 Baselines

We compared performance against traditional item-item similarity measurements:

- *Empirical Conditional Probability (ECP):* Empirical measurement of the conditional probability $P(j|i)$ defined as

$$P(j|i)_{empirical} = \frac{n_{i,j}}{n_i + 1},$$

  where n_i is the total number of occurrences of item i in the dataset, and $n_{i,j}$ is the number co-occurrences of i and j together (counts were smoothed by adding 1).

- *Jaccard Similarity:* Jaccard similarity is a well know similarity measure defined as

$$Jaccard(i,j) = \frac{n_{i,j}}{n_i + n_j - n_{i,j}}.$$

- *Cosine Similarity:* Cosine similarity is another widely used similarity measure defined as

$$Cosine(i,j) = \frac{n_{i,j}}{\sqrt{n_i n_j}}.$$

Note that ranking based on these similarity measures differ in the way of normalizing the co-counts ($n_{i,j}$).

5.4.3 Accuracy Results

We measure performance in terms of Mean Percentile Rank (MPR). This metric was earlier used in studies of implicit feedback datasets within the context of personalized recommendations [49, 50]. In our item-oriented context, it is defined as follows: For each test pair of related items (i, j) we sample N additional random items. We rank every item k (the N random ones and j) with respect to $P(k|i)$ based on our model. Then, we compute the percentile rank of item j within this ranking. Ranks are averaged over all test pairs. Accordingly, percentile ranks closer to zero indicate better rankings. We used $N = 200$ in our experiments, though results are insensitive to changes in N given the large number of test pairs.

We trained the EIR model with $d = 50$ dimensions. Table 5.2 presents the MPR results of EIR against the baselines. We were encouraged by the fact that our algorithm outperformed the baselines on the Netflix, MSD and the Books datasets. On the YMusic dataset, our algorithm was second to ECP with a very small difference between the two.

Figure 5.1 depicts the mean percentile rank of the different algorithms vs. the support (popularity) of the conditioned item i in $P(j|i)$. While all the algorithms perform well on popular items, EIR has a clear advantage in the long tail. This is explained by the fact the EIR employs global optimization that utilizes the entire training data rather than just the pairwise information. Among the different baseline algorithms, ECP is most similar to EIR. This is to be expected, as ECP is merely an empirical estimate of $P(j|i)$ – the objective of EIR.

	EIR	ECP	Jaccard	Cosine
Netflix	**5.825%**	6.57%	9.728%	8.376%
MSD	**6.602%**	13.663%	35.287%	49.18%
YMusic	2.837%	**2.536%**	9.795%	9.545%
Books	**27.342%**	35.902%	53.896%	61.011%

Table 5.2: Comparing MPR of EIR against common baselines (lower is better).

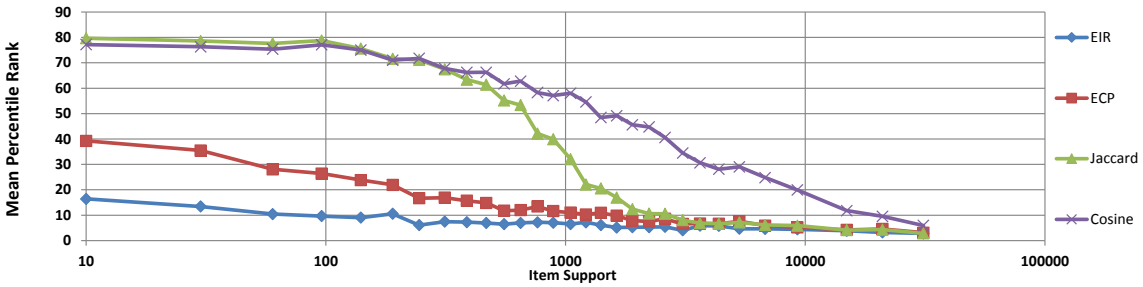


Figure 5.1: Mean Percentile Rank vs. the support of the conditioned item i in $P(j|i)$ in the MSD dataset.

	Netflix	MSD	YMusic	Books
Speedup	25.036	21.422	31.219	11.218

Table 5.3: Speedup values for EIR with 50 dimensions

5.4.4 Fast Retrieval Results

We evaluate the fast retrieval capabilities of EIR using metric trees [36, 51]. Metric trees are binary space-partitioning trees widely used for the task of indexing Euclidean datasets. The tree construction is very fast and space efficient, and the search employs the depth-first branch-and-bound algorithm similar to that of [51]. We quantify the improvement in retrieval time for the task described in (5.7) when using the metric tree compared to a naive search as follows:

$$\text{Speedup} = \frac{\text{Retrieval Time Using Naive Search}}{\text{Retrieval Time Using Tree}}. \tag{5.11}$$

The speedup values for each dataset are presented in Table 5.3.

5.5 Future Work

We plan to extend this work to include also predictions from a set of several items to the next item the user is likely to consume. For example, given a set of items S in a user's virtual shopping cart, we wish to predict the next item i the user is likely to add to her basket, Namely, we wish to model $P(i|S)$. This will better reflect common scenarios of online shopping websites.

VI Additional Work

During my studies I have published few additional papers. This Chapter briefly describes these works.

6.1 Additional Research on Recommender Systems

6.1.1 Real World Recommender Systems

Together with colleagues at Microsoft, we described the original settings of the Xobx Live Recommender system [3]. At the time of publication, the system was based on the Matchbox algorithm [18] which was used to compute personalized games and movies recommendations to over 50 million Xbox users worldwide. Later on, we replaced that system with a novel scalable algorithm designed specifically for one class collaborative filtering [52]. Unlike other algorithms, the algorithm in [52] delineates the odds of a user disliking an item from simply not considering it. We propose a new perspective on the one-class problem which common in most real world settings.

Another publication describes Microsoft's Sage project [53]. Sage is Microsoft's free, all-purpose recommender system, designed and deployed as an ultra-high scale cloud service on Azure. In [53] we give an overview of the systems in terms of algorithms as well as the engineering aspects.

6.1.2 Feature Selection for Recommender Systems

Meta-data features can be used to improve recommendations accuracy and mitigate the cold-start problem. In Chapter 3 we presented a model that uses taxonomy features in order to propagate information between different items sharing the same taxonomy. In our Xbox movies recommender, we used meta-data features in the form of labels to improve our understanding of cold movies [54]. Unlike the taxonomy features, only part of the

movie features are informative or useful with regard to collaborative filtering. In [54] we present a Matrix Factorization model with Embedded Feature Selection (MF-EFS). MF-EFS incorporates a novel sparsity prior on feature parameters to automatically discern and utilize informative features while simultaneously pruning non-informative features.

A different aspect of feature selection for recommendation systems is presented in [55]. In this work we presented a feature selection algorithm designed specifically for collaborative filtering recommender systems. The algorithm can handle different types of meta-data attributes as well as labels. It scores the features according to their informative content with regard to the recommendation task. Feature selection naturally follows by selecting the high scoring features while pruning the low scoring ones.

6.1.3 Group Recommendations

Another paper I published together with colleagues at Microsoft Research Cambridge and Microsoft Research New-York focused around the area of Group Recommendations. In [56] we studied a unique dataset consisting of individual and group television viewing patterns of more than 50 million U.S. viewers in over 50,000 groups atomically recorded by Nielsen[1]. Altogether, we studied a dataset of more than 4 million household views paired with individual level demographic and co-viewing information. Our analysis revealed how engagement in group viewing varies by viewer and content type, and how viewing patterns shift across various group contexts. Furthermore, we leveraged this large-scale dataset to directly estimate how individual preferences are combined in group settings, finding subtle deviations from traditional models of preference aggregation. Finally, we presented a simple model which captures these effects and discuss the impact of these findings on the design of group recommendation systems. To the best of our knowledge, currently this is the largest academic study of group viewing patters and group recommendations to be published.

[1] `www.nielsen.com`

6.2 Music Information Retrieval in Peer-to-Peer Networks

Some of my earlier studies were centered on Music Information Retrieval (MIR) from Peer-to-Peer (P2P) datasets. This chapter briefly summarize this line of work.

6.2.1 Trend Prediction Based on P2P Queries

Detecting Musical Artists Before They Become "Viral"

This line of work deals with detecting musical artists on the Internet right before they become "viral". We monitor the *Word of Mouth* phenomena in file sharing networks in order identify local artists with high probability to be adopted by a much larger audience. In [7, 8], we used geographically identified P2P queries in order to detect local emerging musical talents. The detection algorithm is based on the observation that emerging artists have a discernible stronghold of fans in their hometown area where they are able to perform and market their music. In a file-sharing network, this is reflected as a delta function spatial distribution of content queries. The algorithm mimics human scouts by looking for performers which exhibit a sharp increase in popularity within a small geographic region but low nation-wide popularity.

Predicting Artists' Success on The Billboard Charts

In past decades, air-plays and record sales were the primary means of distribution of popular music. The Billboard Hot 100 was therefore a reasonable proxy to popularity. Today, however, new technologies in particular the Internet, have created new means for distribution of music. The growing popularity of file sharing make record sales and radio plays an increasingly poor predictor of peoples' taste. In [9] we compiled a ranking chart based on P2P queries aiming to serve as an alternative to the Billboard charts. We compared our chart with the official *Billboard Hot 100* chart and show a high correlation (0.89) between the two charts with a time gap of one week in favor of our chart; namely, we detect the same popularity trends shown in the Billboard chart, but one week earlier. Schedl *et al.* continued this line of research in [10], where several alternative artist ranking

charts from various Internet sources were compared.

In a later work [11] we used a dataset of 185.6 million Gnutella queries to predict trends on the *Billboard Hot 100* chart. We employed decision trees algorithms such as C4.5 [57] and BFTree [58, 59] to predict a song's top rank on the Billboard singles chart. We were able to predict accurately songs entrance to the top-10, top-20 and top-30 positions. We also showed ability to predict "flops" – songs that do not make it to the top-50.

6.2.2 Collaborative Filtering Based on P2P Networks

Collecting Datasets from P2P Networks

In [13] we introduced a new collaborative filtering data-set collected from the *Gnutella* network. We employed a Crawler-Browser set-up in order to discover participating nodes and record information on their shared folders content. In [60, 61] we further investigate the content of this dataset and give additional details about the cleaning and post-processing steps.

In addition to the *Gnutella* file sharing network, we experimented with other file sharing networks such as the *Direct Connet (DC)* network. We devised a participating agent for measuring and characterizing the DC file sharing network [12]. Our study, which was the first measurement study of the DC network, discovered a query duplication problem that stems the protocol's scalability potential.

Evaluating the Information Content of P2P Datasets

In this research [62] we compared artist-artist relations derived from our P2P dataset from [13] with other MIR approaches for artist-artist similarity. We compared against CF similarities derived from the Last.fm dataset [63], acoustic models such as Gaussian mixtures over MFCCs and chroma vectors, semantic models such as semantic multinomial auto-tags and social tags from Last.fm[2]), and a text model based on the artists biography data. We evaluate these different data sources against an external partial order artists similarity survey using a similarity prediction task as taken from [64].

[2] http://last.fm/

References

[1] R. M. Bell and Y. Koren, "Lessons from the netflix prize challenge," *SIGKDD Explor. Newsl.*, vol. 9, 2007.

[2] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The yahoo! music dataset and KDD-cup'11," *Journal Of Machine Learning Research*, vol. 18, pp. 3–18, 2012.

[3] N. Koenigstein, N. Nice, U. Paquet, and N. Schleyen, "The xbox recommender system," in *Proc. 6th ACM Conference on Recommender Systems*, 2012.

[4] G. Dror, N. Koenigstein, and Y. Koren, "Web scale media recommendation systems," *Proceedings of the IEEE*, pp. 1–15, 2012.

[5] N. Koenigstein, P. Ram, and Y. Shavitt, "Efficient retrieval of recommendations in a matrix factorization framework," in *CIKM*, 2012.

[6] N. Koenigstein and Y. Koren, "Towards scalable and accurate item-oriented recommendations," in *Proc. 7th ACM Conference on Recommender Systems*, 2013.

[7] N. Koenigstein, Y. Shavitt, and T. Tankel, "Spotting out emerging artists using geo-aware analysis of P2P query strings," in *The 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (Las Vegas, NV, USA), pp. 937–945, 2008.

[8] N. Koenigstein and Y. Shavitt, "Talent scouting in P2P networks," *Computer Networks*, vol. 56, no. 3, pp. 970–982, 2012.

[9] N. Koenigstein and Y. Shavitt, "Song ranking based on piracy in peer-to-peer networks," in *International Symposium on Music Information Retrieval*, (Kobe, Japan), Oct. 2009.

[10] M. Schedl, T. Pohle, N. Koenigstein, and P. Knees, "What's Hot? Estimating Country-Specific Artist Popularity," in *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR 2010)*, (Utrecht, the Netherlands), August 2010.

[11] N. Koenigstein, Y. Shavitt, and N. Zilberman, "Predicting billboard success using data-mining in P2P networks," in *ISM '09: Proceedings of the 2009 11th IEEE International Symposium on Multimedia*, Dec. 2009.

[12] P. Gurvich, N. Koenigstein, and Y. Shavitt, "Analyzing the DC file sharing network," in *IEEE Tenth International Conference on Peer-to-Peer Computing (IEEE P2P)*, pp. 1–4, 2010.

[13] N. Koenigstein, Y. Shavitt, T. Tankel, E. Weinsberg, and U. Weinsberg, "A framework for extracting musical similarities from peer-to-peer networks," in *IEEE International Conference on Multimedia and Expo (ICME 2010)*, 2010.

[14] Y. Koren, R. M. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *IEEE Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[15] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[16] T. K. Moon, "The expectation-maximization algorithm," *IEEE Signal Processing Magazine*, vol. 13, Nov. 1996.

[17] U. Paquet, B. Thomson, and O. Winther, "A hierarchical model for ordinal matrix factorization," *Statistics and Computing*, vol. 21, 2011.

[18] D. H. Stern, R. Herbrich, and T. Graepel, "Matchbox: large scale online bayesian recommendations," in *WWW*, pp. 111–120, 2009.

[19] J. Bennett and S. Lanning, "The netflix prize," in *Proc. KDD Cup and Workshop*, 2007.

[20] P.-L. Chen, C.-T. Tsai, Y.-N. Chen, K.-C. Chou, C.-L. Li, C.-H. Tsai, K.-W. Wu, Y.-C. Chou, C.-Y. Li, W.-S. Lin, S.-H. Yu, R.-B. Chiu, C.-Y. Lin, C.-C. Wang, P.-W. Wang, W.-L. Su, C.-H. Wu, T.-T. Kuo, T. G. McKenzie, Y.-H. Chang, C.-S. Ferng, C.-M. Ni, H.-T. Lin, C.-J. Lin, and S.-D. Lin, "A linear ensemble of individual and blended models for music rating prediction," in *KDD-Cup'11 Workshop*, 2011.

[21] Y. Koren, "Collaborative filtering with temporal dynamics," in *KDD*, pp. 447–456, 2009.

[22] Y. Koren, "The bellkor solution to the netflix grand prize," 2009.

[23] Y. Koren, "Factorization meets the neighborhood: a multifaceted collaborative filtering model," in *The 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 426–434, 2008.

[24] M. Kendall and K. D. Gibbons, *Rank Correlation Methods*. Oxford University Press, 1990.

[25] M. Piotte and M. Chabbert, "The pragmatic theory solution to the netflix grand prize," 2009.

[26] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The Computer Journal*, vol. 7, no. 4, 1965.

[27] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence properties of the nelder-mead simplex algorithm in low dimensions," *SIAM Journal of Optimization*, vol. 9, pp. 112–147, 1996.

[28] M. Wright, "Direct search methods: Once scorned, now respectable," in *Numerical Analysis* (D. Griffiths and G. Watson, eds.), pp. 191–208, Addison Wesley, 1995.

[29] D. Lee and M. Wiswall, "A parallel implementation of the simplex function minimization routine," *Comput. Econ.*, vol. 30, pp. 171–187, 2007.

[30] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pp. 271–280, 2007.

[31] M. Khoshneshin and W. N. Street, "Collaborative filtering via euclidean embedding," in *Proceedings of the fourth ACM conference on Recommender systems*, pp. 87–94, 2010.

[32] K. Clarkson, "Nearest-neighbor searching and metric space dimensions," *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, 2006.

[33] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," in *STOC*, 1998.

[34] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pp. 380–388, 2002.

[35] S. Zhang, W. Wang, J. Ford, and F. Makedon, "Learning from incomplete ratings using non-negative matrix factorization," in *Proceedings of the Sixth SIAM International Conference on Data Mining*, 2006.

[36] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. Springer, 1985.

[37] S. M. Omohundro, "Five Balltree Construction Algorithms," tech. rep., International Computer Science Institute, December 1989.

[38] T. Liu, A. W. Moore, A. G. Gray, and K. Yang, "An Investigation of Practical Approximate Nearest Neighbor Algorithms," in *Advances in Neural Information Processing Systems 17*, 2005.

[39] G. Karypis, "CLUTO a clustering toolkit," tech. rep., Dept. of Computer Science, University of Minnesota, 2002.

[40] G. Linden, B. Smith, and J. York, "Industry report: Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Distributed Systems Online*, vol. 4, no. 1, 2003.

[41] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pp. 285–295, ACM, 2001.

[42] Y. Koren, "Factor in the neighbors: Scalable and accurate collaborative filtering," *TKDD*, vol. 4, no. 1, 2010.

[43] S. Rendle, C. Freudenthaler, and L. Schmidt-Thieme, "Factorizing personalized Markov chains for next-basket recommendation," in *WWW*, pp. 811–820, 2010.

[44] M. Khoshneshin and W. N. Street, "Collaborative filtering via euclidean embedding," in *RecSys*, pp. 87–94, 2010.

[45] Y. Bengio and J.-S. Senécal, "Quick training of probabilistic neural nets by sampling," in *Proc. 9th International Workshop on Artificial Intelligence and Statistics (AISTATS'03)*, 2003.

[46] N. Aizenberg, Y. Koren, and O. Somekh, "Build your own music recommender by modeling internet radio streams," in *WWW*, 2012.

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

[48] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen, "Improving recommendation lists through topic diversification," in *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pp. 22–32, 2005.

[49] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *ICDM*, pp. 263–272, 2008.

[50] H. Steck, "Training and testing of recommender systems on data missing not at random," in *KDD*, pp. 713–722, 2010.

[51] J. K. Uhlmann, "Satisfying general proximity/similarity queries with metric trees," *Inf. Process. Lett.*, vol. 40, no. 4, pp. 175–179, 1991.

[52] U. Paquet and N. Koenigstein, "One-class collaborative filtering with random graphs," in *Proceedings of the 22nd international conference on World Wide Web*, WWW '13, pp. 999–1008, 2013.

[53] R. Ronen, N. Koenigstein, E. Ziklik, M. Sitruk, R. Yaari, and N. Haiby-Weiss, "Sage: Recommender engine as a cloud service," in *Proc. 7th ACM Conference on Recommender Systems*, 2013.

[54] N. Koenigstein and U. Paquet, "Xbox movies recommendations: Variational bayes matrix factorization with embedded feature selection," in *Proc. 7th ACM Conference on Recommender Systems*, 2013.

[55] R. Ronen, N. Koenigstein, E. Ziklik, and N. Nice, "Selecting content-based features for collaborative filtering recommenders," in *Proc. 7th ACM Conference on Recommender Systems*, 2013.

[56] A. J. Chaney, M. Gartrell, J. M. Hofman, J. Guiver, N. Koenigstein, P. Kohli, and U. Paquet, "A large-scale exploration of group viewing patterns," in *TVX - ACM International Conference on Interactive Experiences for Television and Online Video*, 2014.

[57] J. R. Quinlan, "Learning with continuous classes," pp. 343–348, 1992.

[58] H. Shi, "Best-first decision tree learning," Master's thesis, University of Waikato, 2007. COMP594.

[59] J. Friedman, T. Hastie, and R. Tibshirani, "Additive logistic regression : A statistical view of boosting," *Annals of statistics*, vol. 28, no. 2, pp. 337–407, 2000.

[60] N. Koenigstein, Y. Shavitt, E. Weinsberg, and U. Weinsberg, "Measuring the validity of peer-to-peer data for information retrieval applications," *Computer Networks*, vol. 56, no. 3, pp. 1092–1102, 2012.

[61] N. Koenigstein, Y. Shavitt, E. Weinsberg, and U. Weinsberg, "On the applicability of peer-to-peer data in music information retrieval research," in *Proc. 11th International Society for Music Information Retrieval Conference (ISMIR 2010)*, 2010.

[62] N. Koenigstein, G. Lanckriet, B. McFee, and Y. Shavitt, "Collaborative filtering based on P2P networks," in *Proc. 11th International Society for Music Information Retrieval Conference (ISMIR 2010)*, 2010.

[63] O. Celma, *Music Recommendation and Discovery in the Long Tail.* PhD thesis, Universitat Pompeu Fabra, 2008.

[64] B. McFee and G. R. Lanckriet, "Partial order embedding with multple kernels," in *Proc. 26th annual International Conference on Machine Learning (ICML)*, pp. 721–728, 2009.